ON ALGORITHM DESIGN AND PROGRAMMING MODEL FOR MULTI-THREADED COMPUTING

A Dissertation
Presented to
The Academic Faculty

by

Zhengyu He

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2012

ON ALGORITHM DESIGN AND PROGRAMMING MODEL FOR MULTI-THREADED COMPUTING

Approved by:

Professor Bo Hong, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor George Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor David Bader
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Douglas Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: March 15th, 2012

To my parents and wife.

ACKNOWLEDGEMENTS

I know I would not have such a joyful doctoral study without all the support, encouragement, and love I received during the journey. I would like to take this opportunity to express my greatest appreciation to people who have helped and supported me.

My thank first goes to Dr. Bo Hong, my advisor. Dr. Hong introduced me to the richness and depth of parallel computing. Dr. Hong also showed me that creativity and discipline are mutually reinforcing. His sparkling ideas have always inspired me. I am very grateful to Dr. Hong for his guidance, for sharing his wisdom and breadth of knowledge, and for his patience in allowing me to explore.

I would also like to thank my dissertation committee members: Dr. David Bader, Dr. Douglas Blough, Dr. George Riley and Dr. Sudhakar Yalamanchili for their service on my committee and their insightful suggestions and comments on my research.

I also want to thank my proposal committee chair, Dr. Scott Wills, who unfortunately passed away not long ago, for the help and instructions he gave to me.

Next, I want to thank my fellow lab members who provided great collaboration and assistance during my study. I give a big thank to Xiao Yu, Weiming Shi, Jiadong Wu and Chunlei Chen for their support. I also would like to thank all of my friends who make my doctoral study a most pleasant experience.

Last but not least, I owe my Ph.D. degree to my dearest family: my father Pingbo He, my mother Jianjun Wei, and my wife Pingxin Li, for their unfailing love and support through the years. No matter life is up or down, they always stand by my side and provide me all the courage and confidence to face every challenge in my life. No word can express my deepest gratitude for what they have done for me. To them I dedicate this dissertation.

TABLE OF CONTENTS

DEDICATION . . . . . **iii**

ACKNOWLEDGEMENTS . . . . . **iv**

LIST OF TABLES . . . . . **vii**

LIST OF FIGURES . . . . . **viii**

SUMMARY . . . . . **ix**

I INTRODUCTION **1**

II AN ASYNCHRONOUS MULTI-THREADED ALGORITHM FOR MAX-FLOW PROBLEM **5**

2.1 Background and Related Work 6

2.2 The Target Multi-Processor Platform 8

2.3 The Asynchronous Multi-threaded Algorithm 8

2.4 Correctness and Complexity Bound of the Algorithm 12

2.4.1 Proof of Correctness 13

2.4.2 Complexity Bound 21

2.5 Asynchronous Global Relabeling Heuristic 23

2.5.1 Correctness of the AGR heuristic 26

2.6 Programming Implementation of the Algorithm 31

2.6.1 Cache Efficiency 32

2.6.2 Load Balancing 33

2.7 Experimental Results 35

2.8 Summary 42

III PERFORMANCE MODELING OF TRANSACTIONAL MEMORY **45**

3.1 Background and Related Work 47

3.1.1 Summary of TM Systems 48

3.1.2 Performance Modeling of TM 54

3.2 Target TM systems 55
3.3 Analysis of the TM systems 57
3.3.1 Statistical Characteristics of Transactions 57
3.3.2 Impact of Transactional Congestion 58
3.4 Queuing Model of TM systems 59
3.5 Experiments and Discussions 62
3.6 Summary 66
IV ADAPTIVE CONTENTION MANAGEMENT FOR STM SYSTEMS 67
4.1 Background and Related Work 70
4.2 The Necessity of Adaptive Contention Management 73
4.3 Profiling-based Adaptive Contention Management 75
4.3.1 Dynamic Adjustment of Profiling Interval and Profiling Length 78
4.3.2 Profiling Length 79
4.3.3 Selection of Candidate CMs 83
4.4 Implementation 83
4.5 Experimental Results 85
4.5.1 Implementation Overhead 89
4.5.2 Livelock CMs 91
4.5.3 Time-varying Workloads 91
4.6 Summary 92
V CONCLUSION 93
REFERENCES 96

LIST OF TABLES

1 Model input parameters 62

2 Summary of the tested CMs 86

LIST OF FIGURES

1 The impact of the frequency of global relabelling 38

2 Experimental results on Acyclic-Dense graphs. 39

3 Experimental results on Genrmf-long graphs. 40

4 Experimental results on Genrmf-wide graphs. 41

5 Experimental results on Washington-RLG-long graphs. 42

6 Experimental results on Washington-RLG-wide graphs. 42

7 Illustration of transactional execution of the target TM systems. . . . 57

8 Erlang distributions of transaction execution time (TinySTM, eight threads, Redhat 5.4 x86_64, gcc 4.1.2. The bars form the histogram of the execution time, and the solid lines is the fitted curve). 57

9 State Transition Diagram of TM Queuing Model based on CTMC. . . 61

10 Comparison between real executions and theoretical prediction for different benchmarks 62

11 Comparison between real executions and theoretical prediction for different TM implementations 66

12 Impact of different amounts of overhead. 68

13 Comparison of different contention managers on different benchmarks and different platforms. (TinySTM, 16 threads) 74

14 Periodic profiling process of CMs for the proposed adaptive ACM scheme 77

15 A snapshot of the added code of ACM for TinySTM 84

16 Performance comparison of ACM and static CMs on x86 platform for TinySTM. 88

17 Performance comparison of ACM and static CMs on powerpc platform for TinySTM. 89

18 Performance comparison of ACM and static CMs on x86 platform for RSTM. 90

19 Performance comparison of TPM, CPM and APM on the synthetic benchmark (RSTM, x86). 92

SUMMARY

The objective of this work is to investigate the algorithm design and the programming model of multi-threaded computing. With the rapid advancement in multi-/many-core processors, multi-threaded computing has attracted a lot of attention recently. Designing multi-threaded algorithms is very challenging though - when multiple threads need to communicate or coordinate with each other, efficient synchronization support is needed. However, synchronizations are known to be expensive on the emerging multi-/many-core processors, especially when the number of threads increases. To fully unleash the power of such processors, carefully investigations are needed in both algorithm design and programming models for multi-threaded systems.

In this dissertation, we first study the algorithm design aspect of the problem. In particular, we present an asynchronous multi-threaded algorithm for the maximum network flow problem. This algorithm is based on the classical push-relabel algorithm and completely removes the use of locks and barriers from its original parallel version. Experiment results show that such algorithmic innovation can significantly improve the execution speed by enabling higher levels of parallelism.

While this algorithmic method shows effectiveness, it is challenging to generalize the success to other multi-threaded problem. We next focus on improving the synchronization primitives for those algorithms that still need efficient synchronization methods. More specifically, we study the transactional memory, which is believed by many researchers to be a promising mechanism for constructing multi-threaded programs. A queuing-theory-based model is developed to analyze the performance

of different transactional memory systems. Based on the results of the model, we emphasize on the contention management scheme of transactional memory systems that would effectively reduce the contention level and significantly improve the performance. A profiling-based adaptive contention management scheme is proposed to cope with the problem that none of the static contention management schemes can keep good performance on all platforms for all types of workload. Experiment results prove that our adaptive contention management scheme is able to yield a consistently good performance across different benchmarks and platforms.

From the above research, we show that it is necessary and worthwhile to explore both the algorithm design aspect and the programming model aspect for multi-thread computing. New algorithm designs with the target of the multi-threading would benefit from the emerging multi-/many-core platform. At the same time, we still need to improve the current programming model to enhance the programmability and performance for the algorithms that rely on intensive synchronizations.

CHAPTER I

INTRODUCTION

Multi-threaded computing is one of the major forms of parallel computing. It leverages the parallelism by allowing multiple threads to exist within the context of a single process. These threads share the same address space of the process but are able to execute independently. Thus, a key advantage of multi-threaded programs is that they will operate faster on computer systems that have multiple processing cores.

With the recent great development of multi-core processors, multi-threading becomes a *necessary* technique to increase the computing performance when scaling up the frequency is no longer an economic choice. However, except for those embarrassingly parallel problems where no particular effort is needed to segment the problem into a very large number of independent tasks, multi-threading has been considerably more challenging for general applications. This is because multi-threaded algorithms in general need to employ more sophisticated synchronization operations to ensure correctness of results. For example, *locks* or *barriers* are two widely used thread synchronization mechanisms, but they are also known to be difficult to program: underuse of them may cause race conditions; misuse of them may cause deadlocks or livelocks; overuse of them may decrease the performance.

An effective way to alleviate the impact of the synchronization is reducing the needs of synchronization. In this dissertation, we first investigate the algorithmic design aspect of the problem. In particular, we study new algorithms for the maximum network flow problem that has reduced synchronization requirements. The algorithm is based on the classical push-relabel algorithm, which is essentially sequential and

required intensive and costly lock usages to parallelize. The novelty of the algorithm is the complete removal of lock and barrier usages, thereby enabling a much more efficient parallel execution. The newly designed push and relabel operations are executed completely asynchronously and each individual process/thread independently decides when to terminate itself. We further propose an asynchronous global relabeling heuristic to speed up the algorithm. We prove that our algorithm finds a maximum flow with $O(|V|^2|E|)$ operations where the $|V|$ is number of vertices and the $|E|$ is the number of edges in the network. We also prove the correctness of the relabeling heuristic. Extensive experiments show that our algorithm exhibits close-to-linear scalability as the number of processor cores increases and out-performs the lock-based parallel push-relabel algorithm by up to 49% in the execution time.

While the above algorithmic technique proves to be effective, asynchronous algorithm design (to explore thread-level parallelism via reduced synchronization requirement) is very challenging to generalize to arbitrary problems. Under the current situation, a large number of multi-threaded algorithms still need synchronization primitives. To efficiently implement these algorithms, a fast and convenient synchronization method is a necessity. The existing synchronization primitives, such as locks, are notorious for their difficulty to program and debug and their vulnerability to failures and faults [1]. Therefore, in the second half of this dissertation, we focus on optimizing the synchronization primitives. More specifically, we study the transactional memory, a promising replacement for locks. From the angle of transactional memory, every memory read/write access is a transaction to the memory system. The underlying system must ensure the correctness of concurrent accesses, not the programmers. Instead, the programmer only need to locally consider the shared-data access and mark the code accordingly. By using transactional memory, the programmability of multi-threaded programs would be significantly improved.

However, transactional memory also has its disadvantages. While it is still an open

research problem as how transactional memory should be exposed to programmers, the relatively unsatisfactory performance of transactional memory is the major obstacle that prevents practical adoption of this promising technique. Carefully designed parallel programs using lower-level primitives (locks) can outperform their transactional memory versions, often by a substantial margin [2]. Some recent arguments [3,4] even debate whether software transactional memory (STM) can outperform sequential code or not. Thus, improving the performance of transactional memory has been the focus of intensive research activities.

To understand the performance of transactional memory systems better, we first develop a queuing-theory-based analytical model to evaluate the performance of transactional memory. Based on the statistical characteristics observed on actual experiments, we model each transaction as a client requesting services from the computing system. Continuous time Markov chain is used to describe the start and completion (commit or abort) of the transactions. In particular, we study the run-time behavior of transactional memory and analyze the mean transaction execution time to evaluate the performance of target transactional memory systems. Experimental results based on STAMP benchmarks show that our model can predict the performance of real transactional memory systems with an average error rate of 7.9%.

This performance model theoretically reveals that the contention level of transactional memory systems significantly affects their performance. Consequently, we next focus on the contention management (CM) policy in software transactional memory (STM) systems. The CM policy in a STM system decides what action to take when a conflict occurs, and the goal is to reduce the contention level of the STM system by preventing unnecessary conflicts. However, the performance of existing CM policies is sensitive to transaction workload and system platforms. A static policy is therefore unsatisfactory. In this dissertation, we argue that adaptive contention management is necessary and feasible. We further present a profiling-based method that can choose a

suitable CM for a given workload and system platform during run-time. We also propose to use logic-time (transactional commit or abort events) to measure the profiling length and compare it with the traditional physical-time-based method. Experimental results demonstrate that our proposed adaptive contention manager outperforms static CM policies across benchmarks and platforms. In particular, the ACM that uses the number of aborts for the profiling length performs better than others.

The remainder of this dissertation is organized as follows. In Chapter 2, we introduce an asynchronous algorithm for the maximum network flow problem and an asynchronous global relabeling heuristic to speed up the algorithm. Chapter 3 presents a queuing-theory-based analytical model to evaluate the run-time performance of transactional memory systems. In Chapter 4, we describe a profiling-based adaptive CM scheme for STM systems. Finally, in Chapter 5, we conclude the dissertation with discussions and directions for future research.

CHAPTER II

AN ASYNCHRONOUS MULTI-THREADED ALGORITHM FOR MAX-FLOW PROBLEM

To show that a careful algorithm re-design is able to fundamentally improve the synchronization efficiency and, therefore, increase the performance of multi-threaded programs, we present an asynchronous multi-threaded algorithm for max-flow problem in this Chapter. The algorithm is based on the classical push-relabel algorithm, and its original parallel version [5,6] required intensive and costly lock usages. The novelty of the algorithm is in the complete removal of lock and barrier usages, thereby enabling a much more efficient parallel execution. In addition, each thread also independently determines its own termination without using any locks or barriers. Upon the termination of the last thread, the algorithm finds the maximum flow with $O(|V|^2|E|)$ operations. We further develop a non-blocking global relabeling heuristic to speed up the execution of the algorithm. To the best of our knowledge, this is the first completely asynchronous parallel algorithm for the maximum network flow problem.

We implemented our algorithm on x86 platform and compared its performance with the sequential implementation in [7] and the lock-based parallel algorithm in [5]. Three types of graphs were tested: Acyclic-Dense Graph, Washington-RLG Graph, and Genrmf Graph which are taken from the 1^{st} DIMACS Implementation Challenge [8]. The results show that our algorithm outperforms the serial algorithm by up to 89% (50% on average), and outperforms the lock-based algorithm by up to 60% (32% on average). Our algorithm demonstrated better scalability than the lock-based algorithm. Experiments also show that our algorithm completes significantly more (by up to 69%) operations per second than the lock-based algorithm, which

verifies the effectiveness of using asynchronous operations to reduce synchronization overheads. The experiments also demonstrate the effectiveness of the asynchronous global relabeling heuristic.

The rest of this Chapter is organized as follows. Section 2.1 introduces the max-flow problem and summarizes the related works. Section 2.2 presents the model of the target computing platform. The algorithm is presented in Section 2.3 together with the analysis of its asynchronous execution and optimality. Experimental results are shown in Section 2.7. Discussions are provided in Section 2.8.

2.1 Background and Related Work

The maximum network flow (max-flow) problem is a fundamental graph theory problem with important applications in many areas. The problem has attracted extensive research interests over decades.

The max-flow problem is defined as follows: A flow network is a graph $G(V, E)$ where edge $(u, v) \in E$ has capacity c_{uv}. G has source $s \in V$ and sink $t \in V$. A flow in G is a real valued function f defined over $V \times V$ that satisfies the following constraints:

1. $f(u, v) \leq c_{uv}$ for $u, v \in V$
2. $f(v, u) = -f(u, v)$ for $u, v \in V$
3. $\sum_{v \in V} f(v, u) = 0$ for $u \in V - \{s, t\}$

The value of a flow f is defined as $|f| = \sum_{u \in V} f(s, u)$, which is the net amount of flow sent from s to t. The maximum network flow problem searches for a flow with the maximum value.

Both sequential and parallel algorithms have been studied for this problem. Early solutions to the maximum network flow problem are based on the augmenting path method due to Ford and Fulkerson [9], which by itself is pseudo-polynomial and was later improved by carefully choosing the order in which augmenting paths are

selected (e.g. the $O(|V||E|^2)$ algorithm by Edmonds and Karp [10] and the $O(|V|^2|E|)$ algorithm by Dinitz [11]). The concept of preflow was introduced by Karzanov in [12], which leads to an $O(|V|^3)$ algorithm, the execution time was further improved in [13, 14]. Goldberg etc. designed the push-relabel method [15] with $O(|V|^2|E|)$ operations and further improved the complexity bound by using various techniques [7].

In addition to these sequential algorithms, parallel algorithms have also received a lot of attention. For example, the parallel algorithm due to Shiloach and Vishkin [16] runs in $O(|V|^2 \log |V|)$ time using a $|V|-$processor PRAM. Goldberg pointed out that the dynamic-tree-based algorithm in [15] can be implemented on an EREW PRAM, taking $O(|V|^2 \log |V|)$ time and $O(|V|)$ processors. PRAM model [17], however, cannot be considered as a physically realizable model because as the number of processors and the size of the global memory scale up, it quickly becomes impossible to ignore the impact of the interconnection and synchronization overheads.

Practical implementations of parallel algorithms have also been investigated intensively. Anderson and Setubal [5] augmented the push-relabel algorithm with a *global relabeling* operation. Bader etc. [6] designed a parallel implementation using gap relabeling heuristic with considerations in the cache performance of the push-relabel algorithm. Both implementations have demonstrated good execution speed. These parallel implementations, however, share the common feature of using locks to protect every push and relabel operation *in its entirety*, which essentially sequentializes any two push/relabel operations whenever a common vertex is involved. Without lock protection, these implementation will fail to find the maximum flow. Locks are known to have expensive overheads [18]. Parallelism in these algorithms is therefore limited by the intensive lock usages, which can lead to performance degradation especially when the number of processors scales up.

2.2 The Target Multi-Processor Platform

We assume that the target multi-processor platform consists of multiple processing cores that access a shared memory. Symmetric multiple processor (SMP) systems, chip multi-core processors and graphic processing unit (GPU) are examples of such a platform.

We assume that the architecture supports atomic 'read-modify-write' operations, which are supported by most existing multi-core architectures. For instance, on x86-compatible processors, including both Intel and AMD processors, atomic 'read-modify-write' operations can be achieved by adding a 'lock' prefix to the original instruction (e.g. `lock: ADD x, 1`). On Nvidia GPUs supporting CUDA 1.1. or higher, programmers can use the provided atomic functions (e.g. `AtomicAdd()` for 'read-modify-write' operations. When one memory location receives multiple modification requests from different threads, the architecture will automatically sequentialize these modifications and enforce a consistent view for all threads. Thus, we do not needs software locks when only one shared variable needs to be modified atomically. For example, suppose $x \leftarrow x + d_1$ and $x \leftarrow x + d_2$ are executed by two threads simultaneously, we only have to use the atomic 'read-modify-write' instructions/functions, and the architecture will atomically complete one instruction after another. Thus, the final value of x will be the accumulation of d_1 and d_2.

2.3 The Asynchronous Multi-threaded Algorithm

Before presenting the algorithm and its programming implementation, we first briefly re-state some notations for network flow problems.

Given a directed graph $G(V, E)$, function f is called a flow if it satisfies the three constraints in Section 2.1. Given $G(V, E)$ and flow f, the *residual capacity* $c_f(u, v)$ is given by $c_{uv} - f(u, v)$, and the *residual network* of G induced by f is $G_f(V, E_f)$, where $E_f = \{(u, v) | u \in V, v \in V, c_f(u, v) > 0\}$. Thus $(u, v) \in E_f \Leftrightarrow c_f(u, v) > 0$.

For each vertex $u \in V$, $e(u)$ is defined as $e(u) = \sum_{w \in V} f(w, u)$, which is the net flow into vertex u. Constraint 3 in the problem statement requires $e(u) = 0$ for $u \in V - \{s, t\}$. But before our algorithm terminates, we may have $e(u) > 0$ for some vertices (which will turn 0 upon termination of the algorithm). We say vertex $u \in V - \{s, t\}$ is *overflowing* if $e(u) > 0$. When overflowing vertices exist, we call f a pre-flow. An integer valued height function $h(u)$ is also defined for every vertex $u \in V$. We say u is higher than v if $h(u) > h(v)$. We follow the definition in [15] and say h is a valid height function if $(u, v) \in E_f$ implies $h(u) \leq h(v) + 1$. We call $(u, v) \in E_f$ a *regular* residual edge if $h(u) \leq h(v) + 1$. A path in E_f is a regular path if it only consists of regular residual edges. We call $(u, v) \in E_f$ a *special* residual edge if $h(u) > h(v) + 1$.

The algorithm is listed below:

Algorithm 1 The Asynchronous Max-flow Algorithm

1: *Initialize* $h(u)$, $e(u)$, and $f(u, v)$
2: **while** $e(s) + e(t) < 0$ **do**
3: execute applicable *push* or *lift* operations *asynchronously*
4: **end while**

where the **initialize**, **push**, and **lift** operations are defined as follows:

- *Initialize* $h(u)$, $e(u)$, and $f(u, v)$:

  $h(s) \leftarrow |V|$

  $e(s) \leftarrow 0$

  for each $u \in V - \{s\}$

    $h(u) \leftarrow 0$

    $e(u) \leftarrow 0$

  for each $(u, v) \in E$ where $u \neq s$ and $v \neq s$

    $f(u, v) \leftarrow 0$

    $f(v, u) \leftarrow 0$

for each $(s, u) \in E$

$f(s, u) \leftarrow c_{su}$

$f(u, s) \leftarrow -c_{su}$

$e(u) \leftarrow c_{su}$

$e(s) \leftarrow e(s) - c_{su}$

- $Push(u, v')$: applies if u is overflowing, and $\exists v \in V$ s.t. $(u, v) \in E_f$ and $h(u) > h(v)$.

  $v' \leftarrow \text{argmin}_v[h(v) \mid c_f(u, v) > 0 \text{ and } h(u) > h(v)]$

  $d \leftarrow \min[e(u), c_f(u, v')]$

  $f(u, v') \leftarrow f(u, v') + d$

  $f(v', u) \leftarrow f(v', u) - d$

  $e(u) \leftarrow e(u) - d$

  $e(v') \leftarrow e(v') + d$

- $Lift(u)$: applies if u is overflowing, and $h(u) \leq h(v)$ for all $(u, v) \in E_f$,

  $h(u) \leftarrow \min\{h(v) | c_f(u, v) > 0\} + 1$

The algorithm differs from the original push-relabel algorithm in the following two aspects: (1) the push operation sends flow to the *lowest* neighbor in G_f, and (2) the termination condition examines the value of $e(s) + e(t)$ instead of the existence of overflowing vertices. These modifications allow the algorithm to be executed *asynchronously* by multiple threads, which constitutes the major contribution of this Chapter.

The asynchronous execution is better explained through the programming implementation of the algorithm as shown in Program 2. Without loss of generality, we assume that for each vertex $u \in V$ there is one thread responsible of executing

$push(u, v')$ and $lift(u)$. We will use u to denote both vertex u and the thread responsible for vertex u. Note that the number of available threads could be smaller than the number of vertices, in which case one thread will be used for multiple vertices.

After the initialization step, thread u executes the code in Program 2 where e', v', h', h'' and d are per-thread private variables and $h(u)$, $e(u)$, and $c_f(u, v)$ [$c_f(u, v) = c_{uv} - f(u, v)$] are shared among all threads.

Program 2 Program Implementation of Algorithm 1

```
 1: while e(s) + e(t) < 0 do
 2:   if e(u) > 0 then
 3:     e' ← e(u)
 4:     h' ← ∞
 5:     for all (u, v) ∈ E_f do
 6:       h'' ← h(v)
 7:       if h'' < h' then
 8:         v' ← v
 9:         h' ← h''
10:       end if
11:     end for
12:     if h(u) > h' then
13:       d ← min(e', c_f(u, v'))
14:       c_f(u, v') ← c_f(u, v') − d          #executed atomically
15:       c_f(v', u) ← c_f(v', u) + d          #executed atomically
16:       e(u) ← e(u) − d                      #executed atomically
17:       e(v') ← e(v') + d                    #executed atomically
18:     else
19:       h(u) ← h' + 1
20:     end if
21:   end if
22: end while
```

In Program 2, we assume that updates to shared variables $c_f(u, v')$, $c_f(v', u)$, $e(u)$, and $e(v')$ (lines 14-17) are executed atomically by the architecture due to the support of atomic 'read-modify-write' instructions. Line 1 decides whether a thread should terminate or not. Next, the thread finds the lowest neighbor v' for vertex u in E_f (lines 5-11). Based on the height of v', the thread executes either $push(u, v')$ (lines 14-17) or $lift(u)$ (line 19).

While thread u is executing the above code for vertex u, each of the other threads is executing the same code for its own vertex. In our algorithm, the progress at one thread does not need to synchronize with any other threads. Such a property exposes maximum parallelism in the execution of the algorithm. This asynchronous parallel execution is fundamentally different from existing parallel push-relabel algorithms where both u and v need to be locked for $push(u, v)$ and u needs to be locked for $lift(u)$ - without such lock protections these algorithms will fail to find the maximum flow.

2.4 Correctness and Complexity Bound of the Algorithm

In the original push-relabel algorithm, the algorithm terminates when no further push or relabel operations can be applied. However, the absence of applicable push or relabel operations at an individual vertex does not imply the termination, because other vertices may be overflowing. Further more, another vertex may push flow to this idling vertex, making it overflow again. The termination of the algorithm, which becomes true only when there do not exist any applicable push or relabel operations at any vertices, requires a global barrier if implemented in a brute-force manner.

The following lemma shows that our termination condition $e(s) + e(t) < 0$ is equivalent to the existence of applicable push and lift operations.

Lemma 1. *Throughout the execution of the Algorithm 1,* $e(s) + e(t) = 0$ *if and only if there does not exist any overflowing vertices.*

Proof: $f(u, v) = -f(v, u)$ is always maintained throughout the algorithm, which easily leads to $\sum_{u,v \in V} f(u, v) = 0$. Therefore,

$$\begin{aligned}
&\textstyle\sum_{u,v \in V} f(u, v) \\
&= \textstyle\sum_{u \in V} f(u, s) + \sum_{u \in V} f(u, t) + \sum_{u \in V, v \in V - \{s,t\}} f(u, v) \\
&= e(s) + e(t) + \textstyle\sum_{u \in V - \{s,t\}} e(u) \\
&= 0
\end{aligned}$$

Because e(u) is always non-negative for $u \in V - \{s, t\}$ during the execution the algorithm (the only operation that reduces $e(u)$ is $push(u, v)$, which always leaves $e(u)$ non-negative), so it is obvious that as long as there exists any overflowing vertices, we always have $e(s) + e(t) < 0$. Consequently $e(s) + e(t) = 0$ implies that $e(u) = 0$ for all $u \in V - \{s, t\}$. □

Lemma 1 eliminates barrier usages for our algorithm. For any thread u, to determine whether it can terminate or not, it checks the summation of $e(s)$ and $e(t)$ instead of the existence of any overflowing vertices. Even though the two condition are mathematically equivalent, the latter requires a global barrier at all the threads for a successful detection while the former can be examined by each thread individually. Note that $e(s)$ and $e(t)$ are modified only by neighbors of s and t. Consequently, multi-threaded implementation of the algorithm will find $e(s)$ and $e(t)$ to be read-only by most threads, resulting in little contention when updating $e(s)$ and $e(t)$. For those threads that need to update $e(s)$ and $e(t)$, the update will be executed efficiently through the atomic 'read-add-write' instruction supported by the architecture.

2.4.1 Proof of Correctness

We start with the following observations on the algorithm: even though the asynchronous execution at multiple threads may be interleaved in an arbitrary order, the result of the execution actually reduces to that of just two equivalent orders.

We define the '*consequence*' of a $push(u, v')$ to be the values of $e(u)$, $e(v')$, $c_f(u, v')$, and $c_f(v', u)$ after the push, the '*consequence*' of a $lift(u)$ to be the value of $h(u)$ after the lift. We also define the '*trace*' of the interleaved execution of multiple threads to be the order in which instructions from the threads are executed in real time. We say two traces are *equivalent* if they have the same consequences.

The trace of a single push operation can be split into two stages: lines 3-13 and lines 14-17. Lines 3-13 test whether a push is applicable, and if applicable, how much

flow needs to be pushed to which neighbor. We call this the '*preparation*' stage of the push. Lines 14-17 updates the shared variables accordingly, which we call the *action* stage of the push. Similarly, the trace of a single lift operation can also be split into two stages: lines 3-12, and line 19. Lines 3-12 test whether a lift is applicable, and if applicable, what should be the new height of the vertex. This is the '*preparation*' stage of the lift. Line 19 updates the vertex height, which is defined as the '*action*' stage of the lift.

Now we present the following pre-defined traces:

1. a *stage-clean trace* where multiple operations do not have any overlapping in their executions. For example, if a trace contains a push and a lift and the push completes in its entirety before the lift starts, then this is a stage-clean trace.

2. a *stage-stepping trace* where all the operations execute their preparation stages before any one proceeds with its action stage.

With the above notational preparation, we have:

Lemma 2. *Any trace of two push and/or lift operations is equivalent to either a stage-clean trace or a stage-stepping trace.*

The proof of Lemma 2 is straightforward. We simply need to enumerate all the possible pairs of operations that might be interleaved and derive an equivalent trace (either stage-clean or stage-stepping) for each such pair. The detailed proof is omitted here.

It is easy to show that traces with more operations can also be reduced similarly as stated in the next lemma. The proof is similar to that for Lemma 2 and omitted here.

Lemma 3. *For any trace of three or more push and/or lift operations, there exists an equivalent trace consisting of a sequence of non-overlapping traces, each of which is either stage-clean or stage-stepping.*

With Lemmas 2 and 3, we can greatly simplify our discussion by focusing on stage-clean and stage-stepping traces rather than arbitrarily interleaved operations.

Lemma 4. *During the execution of Algorithm 1, if u is an overflowing vertex, then either a push or a $lift$ operation applies to it.*

Proof: If a push operation does not apply to u, we must have $h(u) \leq h(v)$ for all $(u, v) \in E_f$, then $lift(u)$ is applicable. □

Lemma 5. *During the execution of Algorithm 1, vertex height never decreases.*

Proof: only the lift operation can change the height of a vertex. When vertex u is about to be lifted, we have $h(u) \leq h(v)$ for all vertices v such that $(u, v) \in E_f$. So $h(u) < minh(v)|(u, v) \in E_f + 1$, and the lift operation increases $h(u)$. □

Lemma 6. *During the execution of Algorithm 1, there is no regular path (defined in Section 2.3) from the source s to the sink t in the residual network G_f.*

Proof: Assuming for the sake of contradiction that there exists a regular path $p = \{v_0, v_1, \cdots, v_k\}$ from s to t where $v_0 = s$ and $v_k = t$. Without loss of generality, p is a simple path, so $k < |V|$. Because p is a regular path, we have $h(v_i) \leq h(v_{i+1})$ for $i = 0, 1, \cdots, k-1$. Combining these inequalities together, we have $h(s) \leq h(t) + k$. Because $h(t) = 0$, we have $h(s) \leq k < |V|$, which contradicts the fact that $h(s) = |V|$ and is never changed throughout the algorithm. □

Now we examine under what condition special residual edges (defined in Section 2.3) may appear, and what will happen thereafter. Suppose we have two vertices a and b in V. After the initialization step and before any push or lift operations, h is a valid height function. Thus initially (a, b) will be a regular residual edge if $(a, b) \in E_f$, so will (b, a).

If a push or lift operation is executed in its entirety without being interleaved with each other, or if the interleaved execution of multiple operations is equivalent

to a stage-clean trace, then the scenario is the same as the original push-relabel algorithm. For this scenario, h is trivially maintained as a valid height function and all the residual edges remain as regular residual edges.

When we have stage-stepping traces, the situation is more complicated and we discuss below:

Case 1: The execution of $lift(a)$ and $lift(b)$ are interleaved.

Case 1.a: Initially, $(a, b) \in E_f$ and $(b, a) \in E_f$. In this case, we must have $h(a) = h(b)$ because otherwise we either have $h(a) > h(b)$ or $h(b) > h(a)$, then either $push(a, b)$ or $push(b, a)$ can be applied, which contradicts the assumption of this case. For $lift(a)$ to be applicable, we must have $h(c) \geq h(a)$ for all $(a, c) \in E_f$, then $h(a) = h(b)$ implies $h(b) = \min\{h(c)|(a, c) \in E_f\}$ because $(a, b) \in E_f$. So $\min\{h(c)|(a, c) \in E_f\}+1 = h(b)+1 = h(a)+1$ and consequently $lift(a)$ will update $h(a) \leftarrow h(a) + 1$. Similarly, $lift(b)$ will update $h(b) \leftarrow h(b) + 1$. So after the two lift operations, we still have $h(a) = h(b)$. Thus $h(a) \leq h(b) + 1$ is maintained for residual edge (a, b) and $h(b) \leq h(a) + 1$ is maintained for residual edge (b, a). Both (a, b) and (b, a) are still regular residual edges after the two lifts.

Case 1.b: Initially, $(a, b) \in E_f$ but $(b, a) \notin E_f$. In this case, an applicable $lift(a)$ implies $h(a) \leq h(b)$ before the lift because otherwise we need to apply $push(a, b)$ instead. $lift(a)$ updates $h(a) \leftarrow \min\{h(c)|(a, c) \in E_f\} + 1$. Since $(a, b) \in E_f$, $h(b)$ will be polled to compute the min, so the lifted $h(a)$ will be lower than $h(b) + 1$. As $h(b)$ is further increased by $lift(b)$, we must have $h(a) \leq h(b) + 1$ after the two lift operations. (a, b) remains a regular residual edge after the two lifts.

Case 1.c: Initially, $(b, a) \in E_f$ but $(a, b) \notin E_f$. This is symmetric to Case 1.b. Similarly, we will have $h(b) \leq h(a) + 1$ after the two lift operations, and (b, a) remains a regular residual edge.

Case 1.d: Initially, $(a, b) \notin E_f$ and $(b, a) \notin E_f$. This is a trivial case as the two lift operations do not add (a, b) or (b, a) into E_f.

In summary, interleaved execution of $lift(a)$ and $lift(b)$ does not cause either (a, b) or (b, a) to become special residual edges.

Case 2: The execution of $push(a, b)$ is interleaved with $push(b, c)$ where c is different than a and b. It can be shown easily that this particular trace is always equivalent to a stage-clean trace where $push(a, b)$ is executed in its entirety before (or after) $push(b, c)$ is executed in its entirety. Then this case reduces to the original push-relabel algorithm. Consequently h is trivially maintained as a valid height function and the interleaved execution of $push(a, b)$ and $push(b, c)$ does not cause (a, b) or (b, a) to become special residual edges.

Case 3: The executions of $push(a, b)$ and $lift(b)$ are interleaved. According to Lemma 2, this trace is equivalent to either a step-clean or a stage-stepping trace. If it is stage-clean, then the scenario reduces to the original push-relabel algorithm and (a, b) or (b, a) will remain as regular residual edges.

If the trace is equivalent to a stage-stepping trace, we have the following two sub-cases to consider. Note we must have $(a, b) \in E_f$ for $push(a, b)$ to be applicable.

Case 3.a: $(b, a) \in E_f$ before the action stage of $push(a, b)$. In this sub-scenario, $push(a, b)$ may remove (a, b) from E_f and hence remove the requirement that $h(a) \leq h(b)+1$. If $push(a, b)$ does not remove (a, b) from E_f, then $h(a) \leq h(b)+1$ before the push (induction assumption) implies $h(a) \leq h(b) + 1$ thereafter. The operation $lift(b)$ increases $h(b)$ to $\min\{h(w)|(b, w) \in E_f + 1\}$, which implies $h(b) \leq h(a)+1$ after the lift since $(b, a) \in E_f$. Therefore (b, a) remains a regular residual edge.

Case 3.b: $(b, a) \notin E_f$ before the action stage of $push(a, b)$. $push(a, b)$ will add (b, a) into E_f. We have the following two cases to consider:

Case 3.b.i: $(b, a) \in E$. In this case, we must also have $f(b, a) = c_{ba}$ before

the push. Otherwise $f(b,a) \leq c_{ba}$ then we can still push some flow from b to a, which means $(b,a) \in E_f$ - but this contradicts the assumption that $(b,a) \notin E_f$. $push(a,b)$ may remove (a,b) from E_f. Note that the removal of (a,b) does not introduce any special residual edges into E_f.

(b,a) will be added into E_f by the action stage of $push(a,b)$. Note that $lift(b)$ calculates the new height of $h(b)$ during its preparation stage, during which $(b,a) \notin E_f$. So $h(a)$ will not be polled by the preparation stage of $lift(b)$ [i.e. $h(a)$ will not be included when computing $\min\{h(w)|(b,w) \in E_f\}+1$ for $lift(b)$]. Consequently, we may have $h(b) > h(a)+1$ after $lift(b)$ updates $h(b)$. In the mean time, we have $(b,a) \in E_f$ by the end of this trace. The combination of $h(b) > h(a)+1$ and $(b,a) \in E_f$ makes (b,a) a special residual edge in E_f.

When (b,a) becomes a special residual edgel, we have $e(b) > 0$, $(b,a) \in E_f$, and $h(b) > h(a)+1$ after the trace. $h(b) > h(a)+1$ implies a was lower than all of b's neighbors in E_f before the trace (otherwise $h(b)$ would be increased to lower than $h(a)+1$). a being b's lowest neighbor in E_f means $push(b,a)$ is now applicable. Consequently, $lift(b)$ will become applicable only after (1) $h(a)$ is lifted such that $h(a) > h(b)$, or (2) $push(b,a)$ is applied. In (1), $lift(a)$ restores (b,a) as a regular residual edge. Let us examine if $push(b,a)$ is applied as in (2), how much flow will be sent.

Let d denote the amount of flow $push(a,b)$ sends from a to b, and d' denote the amount of flow that $push(b,a)$ will send from b to a. According to the algorithm, $d' = \min\{c_f(b,a), e(b)\}$. $f(b,a) = c_{ba}$ before $push(a,b)$ implies $c_f(b,a) = d$ thereafter. In the mean time, $e(b)$ will be increased by d since $push(a,b)$ just sent d amount of flow to vertex b. Note that $e(b) > 0$ before $push(a,b)$ (otherwise $lift(b)$ will not be applicable), so we have $e(b) > d$ and consequently $d' = \min\{c_f(b,a), e(b)\} = \min\{d, e(b)\} = d$.

$d' = d$ means we will have $f(b,a) = c_{ba}$ upon completion of $push(b,a)$, which removes the special residual edge (b,a) from E_f.

Case 3.b.ii $(b,a) \notin E$. We must have $f(a,b) = 0$ because otherwise $f(a,b) > 0$ leads to $c_f(b,a) = c_{vu} - f(b,a) = 0 + f(b,a) > 0$, which means $(b,a) \in E_f$ and contradicts the assumption that $(b,a) \notin E_f$.

Similar to the previous $(b,a) \in E$ case, we may have $h(b) > h(a) + 1$ when the trace finishes. Because $push(a,b)$ will add (b,a) into E_f, (b,a) thus becomes a special residual edge. Again, similarly to the previous case, a $push(b,a)$ operation becomes immediately applicable when the trace completes. Thus b will not be lifted unless (1) a is lifted such that $h(a) > h(b)$, which restores (b,a) as a regular residual edge, or (2) $lift(b)$ is applied after $push(b,a)$ is applied. For the second case where $push(b,a)$ is applied, because $f(a,b) = 0$ before $push(a,b)$, the same amount of flow sent to b by $push(a,b)$ will be returned to a by $push(b,a)$, which will remove (b,a) from E_f.

Case 4: The executions of $push(a,b)$ are interleaved with $push(c,a)$. This case is symmetric to Case 2, and also reduces to the original push-relabel algorithm. It can be shown easily that interleaved $push(a,b)$ and $push(c,a)$ does not cause (a,b) or (b,a) to become special residual edges.

Case 5: The executions of $push(a,b)$ and $push(c,b)$ are interleaved where c is different than a and b. It is straightforward to show that this particular trace is equivalent to a stage celan trace where $push(a,b)$ is executed in its entirety before (or after) $push(b,c)$ is executed in its entirety. This reduces to the original push-relabel algorithm and does not cause (a,b) or (b,a) to become special residual edges.

Case 6: The executions of $push(a,b)$ and $push(b,a)$ are interleaved. This is impossible because $push(a,b)$ being applicable implies that $h(a) > h(b)$, and thus $push(b,a)$ is inapplicable.

Case 7: The executions of $push(a, b)$ and $lift(a)$ are interleaved. This is impossible because according to Lemma 4, $push(a, b)$ and $lift(a)$ cannot be applicable at the same time.

Case 8: The executions of more than two *push* and $lift$ operations are interleaved. According to Lemma 3, the trace will consists of multiple non-overlapping sub-traces. If all the sub-traces are stage-clean, it reduces to the original push-relabel algorithm. If all the sub-traces are stage-stepping with two operations only, the discussion reduces to Cases 1 - 7. If some of the sub-traces are stage-stepping and have more then two operations, then according to Lemma 4 (that $push(a, b)$ and $lift(a)$ cannot be applicable to vertex a at the same time), we only have the following two scenarios: (1) $push(x_1, y)$,$push(x_2, y)$,$\cdots$, $push(x_k, y)$, and $lift(y)$, (2) $push(x_1, y)$,$push(x_2, y)$,$\cdots$,$push(x_k, y)$, and $push(y, z)$. It can be shown easily that for Scenario 1, the discussion of Case 3 applies to the pair-wise relation between the operations ($push(x_i, y)$ and $lift(y)$); and for Scenario 2, the discussion of Case 2 (or 4) applies to the pair-wise relation between the operations ($push(x_i, y)$ and $push(y, z)$).

In summary, the analysis of the 8 cases above shows that (b, a) may become a special residual edge when $push(a, b)$ and $lift(b)$ are interleaved and $(b, a) \notin E_f$ before the action stage of $push(a, b)$. Once (b, a) becomes a special residual edge, a $push(b, a)$ immediately becomes available, which, upon completion, will remove (b, a) from E_f. If the algorithm terminates, then such following-up push operations must have already been applied (otherwise the algorithm would not terminate) and therefore we do not have any special residual edges upon algorithm completion.

Theorem 1. *If Algorithm 1 terminates, then the pre-flow f it computes is a maximum flow.*

Proof: By Lemma 1, if the algorithm terminates, there do not exist any overflowing vertices. This implies that all the special residual edges, if any, have disappeared - otherwise according to the above analysis there will be applicable push operations.

It is easy to verify that the pre-flow f satisfies constraints 1 and 2 of the max-flow problem throughout the execution of Algorithm 1. If the algorithm terminates, then no vertex is overflowing and f also satisfies constraint 3. So f is a valid flow at the termination of Algorithm 1.

We claim that there is no path from s to t in the residual network $G_f(V, E_f)$ upon completion of Algorithm 1. Suppose for the sake of contradiction there is a path $p = v_0, v_1, \cdots, v_k$ from s to t in G_f where $v_0 = s$ and $v_k = t$. Without loss of generality, this is a simple path so $k \leq |V| - 1$. Because G_f only contains regular residual edges upon completion of Algorithm 1, we have $h(v_i) \leq h(v_{i+1})$ for $i = 0, 1, \cdots, k-1$. Combining these inequailities together, we have $h(s) = h(v_0) \leq h(v_k) + k \leq h(t) + |V| - 1 = |V| - 1$. But $h(s) = |V|$ initially and never changes.

By the max-flow min-cut theorem [19], when there is no path from s to t in the residual network G_f, f is a maximum flow. □

2.4.2 Complexity Bound

To show that Algorithm 1 indeed terminates, We show that the algorithm executes at most $O(|V|^2|E|)$ push/lift operations for a given graph $G(V, E)$.

Lemma 7. *Throughout the execution of Algorithm 1, for any vertex $u \in V - \{s, t\}$, if $e(u) > 0$, then there is a simple path p from u to s in the residual graph, and all the edges along path p are regular residual edges.*

Proof: We prove by constructing such a path.

There may be regular and special residual edges in E_f. As shown by the analysis in the above, (b, a) becomes a special residual edge only when $push(a, b)$ and $lift(b)$ are interleaved and $(b, a) \notin E_f$ before the action stage of $push(a, b)$. Once (b, a) becomes a special residual edge, a $push(b, a)$ immediately becomes available, which, upon completion, will remove (b, a) from E_f. For each such special residual edge, we can perform the corresponding push operation and eliminate it from the residual

graph. We denote the remaining residual edges as E_{fr}. Because the removal of the special residual edges does not add any new residual edges, E_{fr} is a subset of E_f that only consists of regular residual edges. Note that the removal of the special residual does not change any vertex heights.

For an overflowing vertex u, we construct the path from u to s with edges in E_{fr}. Let $U = \{v : \text{there exists a simple path from } u \text{ using edges in } E_{fr}\}$, and suppose for the sake of contradiction that $s \notin U$. Let $\overline{U} = V - U$.

For each pair of vertices $w \in \overline{U}$ and $v \in U$, we must have $f(w, v) \leq 0$, because otherwise $f(w, v) > 0$ implies $f(v, w) < 0$ and hence $c_f(v, w) = c_{vw} - f(v, w) > 0$, which means (v, w) is a residual edge. Hence there exists a simple path of the form $u \rightsquigarrow v \rightarrow w$. But this contradicts the choice of w.

It can be shown easily that

$$\sum_{x \in U} e(x) = \sum_{y \in \overline{U}, z \in U} f(y, z) \leq 0$$

However, $e(x)$ never becomes negative, so we must have $e(x) = 0$ for any vertex $x \in U$. In particular, we have $e(u) = 0$. This contradicts the assumption that u is overflowing. Therefore we must have $s \in U$, which means there exists a simple path from u to s using edges in E_{fr}, which is a subset of E_f that consists of regular residual edges only. □

Lemma 8. *During the execution of Algorithm 1, $h(u) \leq 2|V| - 1$ for all vertices $u \in V$.*

Proof: s and t are never lifted in Algorithm 1. So we always have $h(s) = |V|$ and $h(t) = 0$, both of which are no greater than $2|V| - 1$.

Consider any vertex $u \in V - \{s, t\}$. $h(u) = 0 \leq 2|V| - 1$ initially. $h(u)$ increases for each time $lift(u)$ is applied. Note that $e(u) > 0$ after each $lift(u)$ (otherwise $lift(u)$ will not be applied). According to Lemma 7, there exists a simple path $p = v_0, v_1, \cdots, v_k$ from u to s where $v_0 = u$, $v_k = s$, and $k \leq |V| - 1$, and all

the edges along p are regular residual edges. We therefore have $v_i \leq v_{i+1} + 1$ for $i = 0, 1, \cdots, k-1$. Combining these inequalities together, we have $h(u) = h(v_0) \leq h(v_k) + k = h(s) + k = 2|V| - 1$. □

From this point onward, the complexity analysis is identical to that in [15]. The upper bound on h can be used to further bound the number of lift and push operations. We omit the detailed analysis and present the final theorem directly:

Theorem 2. *Given graph $G(V, E)$ with source s and sink t, the algorithm finds the maximum flow with $O(|V|^2|E|)$ push and lift operations.*

2.5 Asynchronous Global Relabeling Heuristic

Previous studies suggested two heuristics, Global Relabeling and Gap Relabeling, to improve the practical performance of the push-relabel algorithm. The height h of a vertex helps the algorithm to identify the direction to push the flow towards the sink or the source. Global Relabeling heuristic updates the heights of the vertices with their shortest distance to the sink. This can be performed by a backward breadth-first search (BFS) from the sink or the source in the residual graph [7]. The Gap Relabeling heuristic due to Cherkassky also improves the practical performance of the push-relabel method (though not as effective as Global Relabeling [7]). It discovers the overflowing vertices from which the sink is not reachable and then lift these vertices to $|V|$ to avoid unnecessary further operations.

In sequential push-relabel algorithms, Global Relabeling and Gap Relabeling are executed by the same single thread that executes the push and lift operations. Race conditions therefore do not exist. For parallel push-relabel algorithms, the Global Relabeling and Gap Relabeling have been proposed by Anderson [5] and Bader [6] respectively. Both heuristics lock the vertices to avoid race conditions: the global or gap relabeling, push, and lift operations are therefore pair-wise mutually exclusive.

In this Chapter, we develop a new Asynchronous Global Relabeling (AGR) heuristic to speed up our asynchronous algorithm. The execution of our heuristic can be arbitrarily interleaved with the push operations, which is fundamentally different from the existing parallel relabeling heuristics. The AGR heuristic is listed in Program 3. Due to the presence of the AGR heuristic, the push and lift operations are also updated as shown in Program 4. In Programs 3 and 4, $color[v]$ (for $v \in V$), *CurrentWave*, and *CurrentLevel* are private variables of the AGR thread. The AGR thread also maintains a private queue for the BFS traversal. $h(v)$, $w(v)$, and $c_f(u, v)$ are shared across all the threads.

With the AGR heuristic, the algorithm dedicates one thread to the execution of the heuristic while other threads simultaneously execute the push and lift operations. To amortize the computational cost, the AGR heuristic is applied periodically after a certain number of push and lift operations.

It was pointed out in [7] that the most accurate height to globally relabel vertex v should be $min(h(v,t), h(v,s) + |V|)$, where $h(v,t)$ and $h(v,s)$ denote the shortest distances from v to the sink and the source respectively. In our global relabeling heuristic, a backwards BFS from the sink is first performed (lines 6-20). If the generated BFS tree does not cover all the vertices in the graph (line 21), the remaining vertices must be disconnected from the sink and their $h(v,t)$ are therefore ∞. Another backwards BFS from the source is then performed to scan the remaining vertices (lines 22-38).

The AGR heuristic also assigns a wave number $w(u)$ to each vertex u. $w(u)$ represents the number of times u has been globally relabeled. The updated push operation requires flow to be pushed within the same wave, or from an older wave to a newer wave (line 14).

Because the global relabeling heuristic and the lift operations are executed by separated threads, they may update the height of the same vertex simultaneously

Program 3 The Asynchronous Global Relabeling Heuristic

```
 1: CurrentWave ← CurrentWave + 1
 2: for all vertex v ∈ V do
 3:    color[v] ← 0
 4: end for
 5: Enqueue the sink
 6: CurrentLevel ← 0
 7: while queue ≠ ∅ do
 8:    Dequeue a vertex u
 9:    CurrentLevel ← CurrentLevel + 1
10:    for all vertex v|(v, u) ∈ E_f do
11:       if color[v] = 0 then
12:          color[v] ← 1
13:          if h(v) < CurrentLevel then
14:             h(v) ← CurrentLevel
15:          end if
16:          w(v) ← CurrentWave
17:          Enqueue vertex v
18:       end if
19:    end for
20: end while
21: if Not all the vertices are colored then
22:    Enqueue the source
23:    CurrentLevel ← |V|
24:    while queue ≠ ∅ do
25:       Dequeue a vertex u
26:       CurrentLevel ← CurrentLevel + 1
27:       for all vertex v|(v, u) ∈ E_f do
28:          if color[v] = 0 then
29:             color[v] ← 1
30:             if h(v) < CurrentLevel then
31:                h(v) ← CurrentLevel
32:             end if
33:             w(v) ← CurrentWave
34:             Enqueue vertex v
35:          end if
36:       end for
37:    end while
38: end if
```

(lines 14 and 31 of Program 3 and line 18 of Program 4). To prevent this from happening, we rely on vertex allocation (Section 2.6.2) to guarantee that a vertex can be either globally relabelled or lifted, but not both. More precisely, our vertex

Program 4 Updated push and lift due to the AGR Heuristic

```
1: while e(s) + e(t) < 0 do
2:   if e(u) > 0 then
3:     e' ← e(u)
4:     h' ← ∞
5:     for all (u, v) ∈ E_f do
6:       h'' ← h(v)
7:       if h'' < h' then
8:         v' ← v
9:         h' ← h''
10:      end if
11:    end for
12:    if h(u) > h' then
13:      d ← min(e', c_f(u, v'))
14:      if w(u) ≤ w(v) then
15:        if h(u) > h(v) then
16:          c_f(u, v') ← c_f(u, v') − d                    #atomic
17:          c_f(v', u) ← c_f(v', u) + d                    #atomic
18:          e(u) ← e(u) − d                                #atomic
19:          e(v') ← e(v') + d                              #atomic
20:        end if
21:      end if
22:    else
23:      if h(u) < h' + 1 then
24:        h(u) ← h' + 1
25:      end if
26:    end if
27:  end if
28: end while
```

allocation strategy guarantees that lines 13-14 (and 30-31) of Program 3 and lines 23-24 of Program 4 are mutually exclusive. Note that our algorithm allows the push operations to be arbitrarily interleaved with either the AGR heuristic or the lift operations.

2.5.1 Correctness of the AGR heuristic

We prove that the AGR heuristic (with the updated push and lift operations in Program 4) computes a maximum flow for any given graph $G(V, E)$ with $O(|V|^2|E|)$ push and lift operations.

We first examine the interleaved execution of the AGR heuristic and the push operations. Similar to Program 2, each push and lift operation in Program 4 is performed in two stages. For the push operation, lines 2-15 constitute the preparation stage, determining whether a push is applicable, and the amount of flow to be pushed if it is applicable; lines 16-19 constitute the action stage that actually performs the push. For the lift operation, the preparation stage consists of lines 2-12, the action stage consists of lines 23-25. The AGR heuristic also has two stages:

1. the preparation stage: lines 9-11 (26-28) determines whether $h(v)$ needs to be updated and the new value of $h(v)$ if it needs to be updated.

2. the action stage: lines 13-16 (30-33) updates $h(v)$ to the new value, and $w(v)$ to the current wave number.

Because our algorithm does not lock the vertices, the two stages of the AGR heuristic may be arbitrarily interleaved with the push operations. We can extend the definition of traces to include the AGR heuristic, and show that all traces (of interleaved AGR heuristic, push, and lift) are equivalent to either stage-clean or stage-stepping traces. The formal statement and the proof of the lemma are similar to that of Lemmas 2 and 3 and omitted here.

Similar to Algorithm 1, the interleaved execution of the AGR heuristic and the push operations may lead to special residual edges. Let us examine under what condition such interleaved execution may cause special residual edges, and what will happen thereafter. We use $async\_global\_relabel(u)$ to denote the global relabeling operation for vertex u.

If the execution of $async\_global\_relabel(u)$ is not interleaved with any push operations, the correctness of the AGR heuristic is the same as that in [5] (which locks the vertices to prevent interleaved executions).

Next we consider the following two interleaving scenarios:

- A stage-stepping trace consists of $push(u, v)$ and $async\_global\_relabel(v)$.

$push(u, v)$ may add (v, u) into E_f. We claim the newly added (v, u) will be a regular residual edge.

According to line 14 of Program 4, we have $w(u) \leq w(v)$ for $push(u, v)$ to be applicable. Actually, we must have $w(u) = w(v)$ because $w(u) < w(v)$ would imply that v has already been relabeled and thus we should relabel u instead.

The action of $async\_global\_relabel(v)$ increases $w(v)$ by 1, which results in $w(u) \neq w(v)$ and temporarily removes the constraint that $h(v) \leq h(u) + 1$.

When u is relabeled later, $w(u)$ will be increased to the same value of $w(v)$, and will thus bring back the constraint that $h(v) \leq h(u) + 1$. This constraint is trivially satisfied: because u is relabeled later, according to the BFS order of the AGR heuristic, we will have $h(u) \geq h(v)$ after u is relabeled, which satisfies $h(v) \leq h(v)+1$, and (v, u) is thus a regular residual edge.

- A stage-stepping trace consists of $push(u, v)$ and $async\_global\_relabel(u)$.

$push(u, v)$ may add (v, u) into E_f, which may or may not be a special residual edge. We have two sub-cases:

Case 1: $w(u) < w(v)$ before the action of $push(u, v)$. This implies that v has already been relabeled and $async\_global\_relabel(u)$ will increase $w(u)$ such that $w(u) = w(v)$, which will bring back the constraint that $h(v) \leq h(u)+1$. Due to the BFS order that the AGR relabels the vertices, u will not be relabeled lower than v because u is relabeled later than v. We therefore will have $h(u) \geq h(v)$ when $async\_global\_relabel(v)$ completes, which trivially satisfies $h(v) \leq h(u)+1$, and thus (v, u) is a regular residual edge in this case.

Case 2: $w(u) = w(v)$ before the action of $push(u, v)$. $w(u) = w(v)$ implies that neither u or v has been relabeled, which means u will be relabeled earlier than v. The action of $async\_global\_relabel(u)$ will lead to $w(u) > w(v)$, and temporarily remove the height constraint between $h(u)$ and $h(v)$. But the constraint will re-appear after

async_global_relabel(v), as it will increase $w(v)$ such that $w(u)$ is equal to $w(v)$ again. Therefore we need to examine the status of the vertices after *async_global_relabel*(v).

Case 2.a: $(v, u) \in E_f$ before the action of *push*(u, v). The existence of (v, u) will cause the AGR heuristic to add v as a neighbor of u and subsequently relabel $h(v)$ to $h(u)+1$. Therefore we have $h(v) \leq h(u)+1$ after *async_global_relabel*(v) completes, and (v, u) is a regular residual edge.

Case 2.b: If $(v, u) \notin E_f$ before the action of *push*(u, v). In this case, we must also have $f(v, u) = c_{vu}$ before the push. Otherwise $f(v, u) < c_{vu}$ implies $c_f(v, u) = c_{vu} - f(v, u) > 0$ and subsequently $(v, u) \in E_f$, which contradicts the assumption that $(v, u) \notin E_f$.

Case 2.b.i The action of *push*(u, v) completes before the preparation of *async_global_relabel*(v) starts. *push*(u, v) will add (v, u) into E_f, which will cause the AGR heuristic to add v as a neighbor of u and subsequently relabel $h(v)$ to $h(u) + 1$. Therefore we have $h(v) \leq h(u) + 1$ for $(v, u) \in E_f$ after *async_global_relabel*(v) completes, and (v, u) is a regular residual edge in this case.

Case 2.b.ii The action of *push*(u, v) completes after the preparation of *async_global_relabel*(v) completes. (v, u) does not exist in E_f during the preparation of *async_global_relabel*(v), therefore v is not considered as a neighbor of u and the relabeling may result in $h(v) > h(u) + 1$, which will make (v, u) a special residual edge once *push*(u, v) completes (and thus adds (v, u) into E_f).

When (v, u) becomes a special residual edge, at the same time, u becomes the lowest neighbor of v in E_f having the same wave number (other neighbors of v either have been relabeled to $h(v) - 1$, or have not been relabeled yet and thus having smaller wave numbers), thus a new *push*(v, u) operation is now applicable. Next we examine how much flow *push*(v, u) will send.

Let d' denote the amount of flow $push(v,u)$ will send from v to u. According to the algorithm, $d' = \min\{c_f(v,u), e(v)\}$. $f(v,u) = c_{vu}$ before $push(u,v)$ implies $c_f(v,u) = d$ thereafter (where d is the amount of flow that $push(u,v)$ sends from u to v). $push(u,v)$ will increase $e(v)$ by d. Note that $e(v) \geq 0$ before $push(u,v)$, we will have $e(v) \geq d$ after the push. Consequently, $d' = \min\{c_f(v,u), e(v)\} = \min\{d, e(b)\} = d$.

$d' = d$ means we will have $f(v,u) = c_{vu}$ upon completion of $push(v,u)$. Thus the special residual edge (v,u) will be removed from E_f.

The above analysis shows that a special sequence of interleaved push and global relabel operations can cause special residual edges. Once (v,u) becomes a special residual edge under this situation, a $push(v,u)$ immediately becomes available, which, upon completion, will remove (v,u) from E_f. Note that The AGR heuristic trivially terminates after it sweeps through all the vertices, by which time all the vertices will have the same wave number.

Because there will not be any overflowing vertices when the algorithm terminates, this implies that all the special residual edges caused by the interleaved push and global relabeling operations will be disappear - otherwise according to the above analysis there will be applicable push operations. We therefore have the following Theorem:

Theorem 3. *With the AGR heuristic and the updated push and lift operations, if Algorithm 1 terminates, then the pre-flow f it computes is a maximum flow.*

The proof is similar to that of Theorem 1 and hence omitted here.

Further more, the above analysis leads to the following lemma:

Lemma 9. *Upon completion of the AGR heuristic, for any vertex $u \in V - \{s,t\}$, if $e(u) > 0$, then there is a simple path p from u to s in the residual graph, and all the edges along path p are regular residual edges.*

Proof Sketch: The above analysis shows that the special residual edges caused by interleaved push and global relabel operations can be removed from the residual graph E_f without changing the height of any vertices. Further more, the remaining residual graph, which is a subset of the E_f that consists of regular residual edges only, must contain a simple path from u to s. The detailed proof of this lemma is similar to that of Lemma 7 and omitted here. Note that we require the AGR heuristic and the lift operations to be mutually exclusive, so we do not need to consider the situation where push, lift, and global relabeling operations are interleaved. □

Lemma 9 leads to the following lemma:

Lemma 10. *Upon completion of the AGR heuristic,* $h(u) \leq 2|V| - 1$ *for all vertices* $u \in V$.

The proof of Lemma 10 is similar to that of Lemma 8 and omitted here. This upper bound on vertex height further bounds the total number of push and lift operations to $O(|V|^2|E|)$, which means that Algorithm 1, when augmented with the AGR heuristic, still has the same complexity bound as the original Algorithm 1. This is stated in the following theorem:

Theorem 4. *With the AGR heuristic and the updated push and lift operations, Algorithm 1 finds a maximum flow with* $O(|V|^2|E|)$ *push and lift opeations for any given graph* $G(V, E)$ *with source* s *and sink* t.

The proof of Theorem 4 is similar to Theorem 2: the upper bound on h is used to limit the number of push and lift operations. Details of the proof is omitted here.

2.6 Programming Implementation of the Algorithm

To validate the efficiency of the proposed asynchronous algorithm, we implemented the algorithm using C and the pthread library. In this section, we discuss the major programming techniques used to accelerate the implementation.

2.6.1 Cache Efficiency

The memory allocation of the vertices and edges is important for the cache performance and hence the practical execution speed of our implementation. In our implementation, the variables associated with every vertex (height, excess flow, wave number, etc.) are packed in a C struct and allocated continuously in the memory for improved locality. The edge locality, however, is more complicated.

Historically, there have been two edge allocation schemes. In the first scheme, it is observed that during the execution of the algorithm, an edge $(u, v) \in E$ in the original graph may induce two edges (u, v) and (v, u) in the residual graph E_f. Flow may exist along both edges. In particular, The action stage of $push(u, v)$ needs to increase the flow along one edge, and decrease the flow along the other. Since the pair of edges are always updated together, this scheme packs the two edges contiguously in the memory (e.g. into a single C struct). This scheme was proposed in [20] and also used in [6]. The other scheme sorts the edges leaving the same vertex and allocates these edges contiguously in the memory. In this scheme, (u, v) and (v, u) most likely cannot reside in the memory side by side. Pointers are thus used to expedite accesses to such edges that are updated in pairs. This scheme is used in the code published by [7].

We experimented both schemes in our implementation. The second scheme, which allocates outgoing edges continuously, slightly outperforms the first one. We conjecture the reason is that each push operation in our algorithm needs to search for the lowest neighbor during the preparation stage, while existing algorithms only need to find any neighbor that is lower by 1. The search procedure requires our algorithm to read all the outgoing edges of a vertex. The second edge allocation scheme therefore improves the locality for the preparation stage of pushes at the cost of affecting the action stage. Because the preparation stages in general need to access more edges than the action stage, the second scheme explores the trade-offs better. The scheme

also improves the execution of the lift operation: the preparation stage of a lift operation needs to access all the outgoing edges of a vertex and will benefit from the improved data locality. Based on these reasons, we adopt the second scheme of edge layout in our implementation.

2.6.2 Load Balancing

Our algorithm can be implemented with an arbitrary number of threads. To achieve meaningful acceleration on a multicore or a multi-processor system, we need to limit the number of threads to the number of hardware contexts that the system supports. Under this practical limit, each thread needs to be in charge of multiple vertices. Load balancing is therefore crucial to the efficiency of our algorithm. The key to load balancing is the allocation of overflowing vertices to keep all the threads busy.

Previous programming implementations use a global queue to maintain the overflowing vertices. When a thread finishes processing the overflowing vertices that were obtained from the global queue, all the newly generated overflowing vertices (by push operations) will be added to the global queue, from which other threads can request a subset of such vertices to work on. The process continues until none of the vertex overflows. Because the global queue is accessed by all the threads and thus needs to be protected (locked) for concurrent accesses, it is likely to become a performance bottleneck especially as the number of threads increases.

We hereby introduce our strategy of using distributed queues to balance the load. Every thread maintains a local queue for overflowing vertices. Since these are local queues, no lock is required to protect their accesses. Every thread processes its local vertex queue in FIFO order. In our scheme, each thread processes the vertices in the local queue and insert all the newly generated overflowing vertices back to the local queue. Communication between threads only occurs when a thread empties its local queue (and hence needs to request overflowing vertices from other threads).

When a thread T1 empties its local queue, it will search and find another thread T2 that has extra overflowing vertices. T1 will send a request by atomically setting a flag *Exchange* of T2 if it is not set. After successfully setting T2's *Exchange*, T1 will lock the its own *Exchange*. With the two flags, other threads will not attempt to send any requests to T1 or T2 until the vertex exchange between T1 and T2 completes. This *Exchange* flag is similar to a lock but it contains the sender thread's ID. T2 will notice (after finishing the current push or lift operation) that T1 has requested overflowing vertices. T2 will split its local queue and give a certain number of vertices to T1. T2 will then reset the *Exchange* flags for T1 and itself, thereby completing the vertex exchange.

The above load balance scheme has the advantage of allowing a thread to locally execute as much operations as possible before exchanging vertices with other threads. In addition, each exchange involves only two threads so that other threads are not affected during the process. Additionally, the elimination of the global queue also reduces the memory footprint of the program.

To prevent a vertex from being lifted and globally relabeled by two threads simultaneously, we need to protect the update of the vertex height (lines 13-15/30-32 of Program 3 and lines 23-25 of Program 4). In our implementation, we do not need to stall any threads in case of such a conflict. For the lift operations, we let them skip the vertex that is currently being relabeled and process the remaining vertices in the local queue. For the global relabel operations, we let them skip the vertex that is currently being lifted and process the following vertices at the same BFS level. The skipped vertex will be reexamined later: (1) our algorithm iterates through the vertices until it terminates so a skipped push operation will be reexamined, and (2) for the global relabeling operation, because it is BFS, it needs to complete vertices at one level before moving to the next level. So if a vertex is skipped, it will be revisited

before the heuristics moves to the next level of vertices. Our implementation is therefore non-blocking. We use the atomic compare-and-swap instruction to determine whether we should continue with the relabel/lift operation or skip to the next vertex.

2.7 Experimental Results

To evaluate the performance of the proposed asynchronous algorithm, a multi-threaded implementation was developed using C and the pthread library. The atomic read-modify-read operation in our algorithm was implemented by using atomic fetch-and-add instructions supported by the x86 architectures. The sequential push-relabel by Goldberg provides the baseline for performance evaluation. Note that the algorithm in [7] only uses the push-relabel method to find out the value of the maximum flow value and then construct a valid maximum flow using the preflow method, which is of $Q(|V||E|)$ complexity. We also compared the performance of our algorithm against multi-threaded lock-based push-relabel algorithms described in [5].

The following programs were implemented for the experiments:

1. **amf**: Our multi-threaded asynchronous push-relabel algorithm with asynchronous global relabeling. FIFO order is maintained for each local queue. As proved in Section 2.4.2, our algorithm is of $O(|V|^2|E|)$ complexity.

2. **lmf**: The lock-based implementation of multi-threaded push-relabel algorithm with concurrent global relabeling due to Anderson [5]. Each thread processes its vertices in a FIFO order. This algorithm is also of $O(|V|^2|E|)$ complexity.

3. **q_prf**: The sequential push-relabel algorithm due to Goldberg [7], with global relabeling. Vertices are processed in FIFO order. The complexity of this algorithm is $O(|V|^3)$ [7].

4. **hi_pr**: The sequential push-relabel algorithm with global relabeling and gap relabeling. In this implementation, vertices are processed in the descending

order of their heights, which leads to the complexity of $O(|V|^2\sqrt{|E|})$ [21]. This is currently the fastest sequential implementation of push-relabel algorithm that we are aware of.

We conducted experiments on both Intel and AMD platforms. The Intel platform has four Six-Core Xeon E7450 processors running at 2.4GHz with 64GB DDR2 800MHz memory. The AMD platform has four 64-bit Quad-Core Opteron 8358 processors running at 2.4GHz with 64GB DDR2 667Hz memory. The operating system (Redhat Enterprise 5 distribution) runs Linux kernel version 2.6.18 and gcc version 4.1.2 was used to generate the executables.

Five types of graphs were used in the experiments. These graphs were used in the 1^{st} DIMACS Implementation Challenge [8]. They represented a variety of graph topologies and sizes and were used to test existing max-flow algorithms [6].

1. **Acyclic-Dense Graphs**: These are complete directed acyclic dense graphs: each vertex is connected to every other vertex. We tested graphs of 2000 and 4000 vertices.

2. **Genrmf-long graphs**: These graphs are comprised of l_1 square grids of vertices (frames) each having $l_2 \times l_2$ vertices. The source vertex is at a corner of the first frame, and the sink vertex is at the opposite corner of the last frame. Each vertex is connected to its grid neighbors within the frame and to one vertex randomly chosen from the next frame. We tested the graphs of $l_1 = 256, l_2 = 32$ (262144 vertices and 1276928 edges) and $l_1 = 512, l_2 = 32$ (524288 vertices and 2554880 edges).

3. **Genrmf-wide graphs**: The topology is the same as genrmf-long graphs except for the values of l_1 and l_2. Frames are bigger in Genrmp-wide graphs than in Genrmp-long graphs. We tested the graphs of $l_1 = 64, l_2 = 64$ (262144 vertices and 1290240 edges) and $l_1 = 128, l_2 = 64$ (524288 vertices and 2584576 edges).

4. **Washington-RLG-long graphs**: These graphs are rectangular grids of vertices with w rows and l columns. Every vertex in a row has three edges connecting to random vertices in the next row. The source and the sink are external to the grid, the source has edges to all vertices in the top row, and all vertices in the bottom row have edges to the sink. We tested the graphs of $w = 256, l = 512$ (131074 vertices and 392960 edges) and $w = 256, l = 768$ (196610 vertices and 589568 edges).

5. **Washington-RLG-wide graphs**: Same as Washington-RLG-long graphs except for the values of w and l. Each row in the Washington-RLG-wide graphs are wider. We tested the graphs of $w = 512, l = 256$ (131074 vertices and 392704 edges) and $w = 768, l = 256$ (196610 vertices and 589056 edges)

We first evaluated the impact of global relabeling. When the AGR heuristic is applied, the execution time is greatly reduced for all the input graphs, by as much as 500 times. The frequency of applying the AGR heuristic also affects the execution time. The impact on the Genrmf-long ($l_1 = 256, l_2 = 32$) graphs is shown in Figure 1 (we observed the same trend on other graphs and the results are omitted here). Figure 1 shows that, when the AGR heuristic is applied more frequently (from every $4|V|$ to $2|V|$, $|V|$ and $|V|/2$ push/lift operations), the execution time reduces, though the improvement is marginal beyond $2|V|$. In the experiments, we also observed (not shown in Figure 1) that execution time increased when the frequency of global relabeling is further increased. In summary, the experiments suggested that applying the AGR heuristic after about $|V|$ push/lift operations is a reasonable choice.

We then compare the performance of the four grams on the five input graph topologies. For each type of graphs, 50 instances were generated using different seeds for the pseudo-random generator. For each program, each instance was tested 3 times. The execution results reported for each program were the averages over the 150 runs.

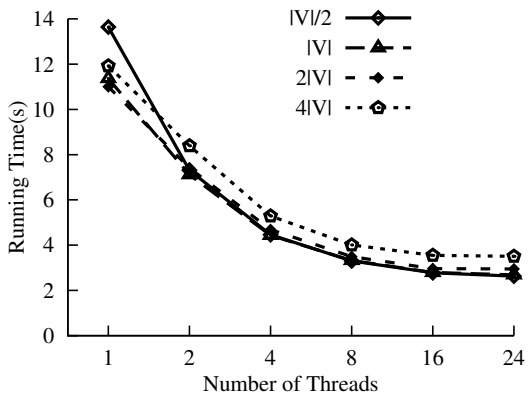
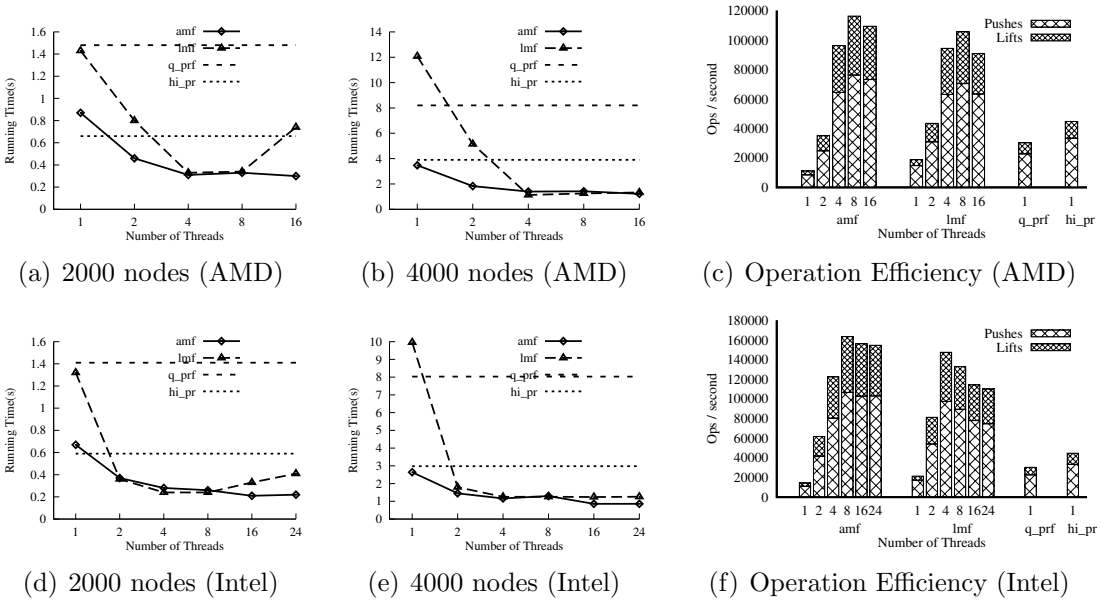


Figure 1: The impact of the frequency of global relabelling

For each set of experiments, we report the execution time of the four programs. However, execution time by itself does not describe all the aspects of a parallel push-relabel algorithm. Different vertex ordering schemes lead to different number of operations, which in turns depends on the input graphs. For example, we observed that on acyclic dense graphs, **hi_pr** executed more operations than our **amf** algorithm but less operations than **q_prf**. Furthermore, due to concurrent executions at multiple threads, parallel algorithm (both **amf** and **lmf**) cannot keep strict FIFO orders. Consequently, sequential algorithms may execute more operations than the sequential algorithms.

Consequently, in order to demonstrate the benefit of lock-free synchronizations, we also evaluated the four programs in terms of their operation efficiency, which is calculated as the average number of push and lift operations that a thread executes in one unit of time.

Figure 2 shows the experimental results on acyclic dense graphs. For dense graphs, the **q_prf** algorithm and the the **hi_pr** algorithm have effectively the same complexity bound ($O(|V|^3) \simeq O(|V|^2\sqrt{|E|})$), and both are lower than that of the **amf** and **lmf** algorithms ($O(|V|^2|E|)$). But when multiple threads were used, the parallel algorithms were able to outperform the sequential algorithms (except for a few cases



(a) 2000 nodes (AMD) (b) 4000 nodes (AMD) (c) Operation Efficiency (AMD)

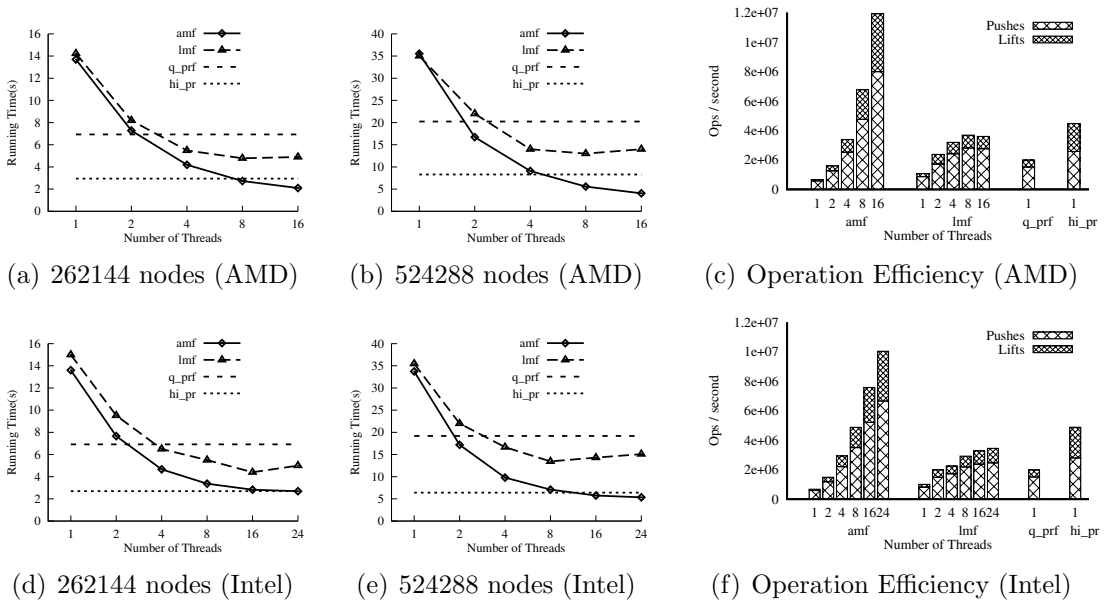
(d) 2000 nodes (Intel) (e) 4000 nodes (Intel) (f) Operation Efficiency (Intel)

Figure 2: Experimental results on Acyclic-Dense graphs.

with the **lmf** algorithm). This demonstrated that parallel algorithm works well for such input graphs.

Figure 2 also shows that our **amf** algorithm outperforms the **lmf** algorithm in terms of both execution time and operation efficiency. Figure 2 (c) and (f) show that our **amf** algorithm executed 21% more operations per second than the **lmf** algorithm. This demonstrated the superiority of our algorithm in executing the individual push/lift operations asynchronously.

Furthermore, we also observed that as the number of threads increases to larger than 8, the execution time of the **lmf** algorithm increased for graphs with 2000 vertices (from 0.34s to 0.74s on the AMD system with 16 threads, and from 0.24s to 0.33s on the Intel system with 16 threads). We conjecture this is due to the deteriorated lock contentions caused by the large number of threads. On the contrary, our algorithm outperforms the **q_prf** and **hi_pr** algorithms whenever more than 2 threads were used, and the execution time reduced each time when we increased the number of threads. When the graphs have 4000 vertices, the threads were less likely to compete



(a) 262144 nodes (AMD) (b) 524288 nodes (AMD) (c) Operation Efficiency (AMD)

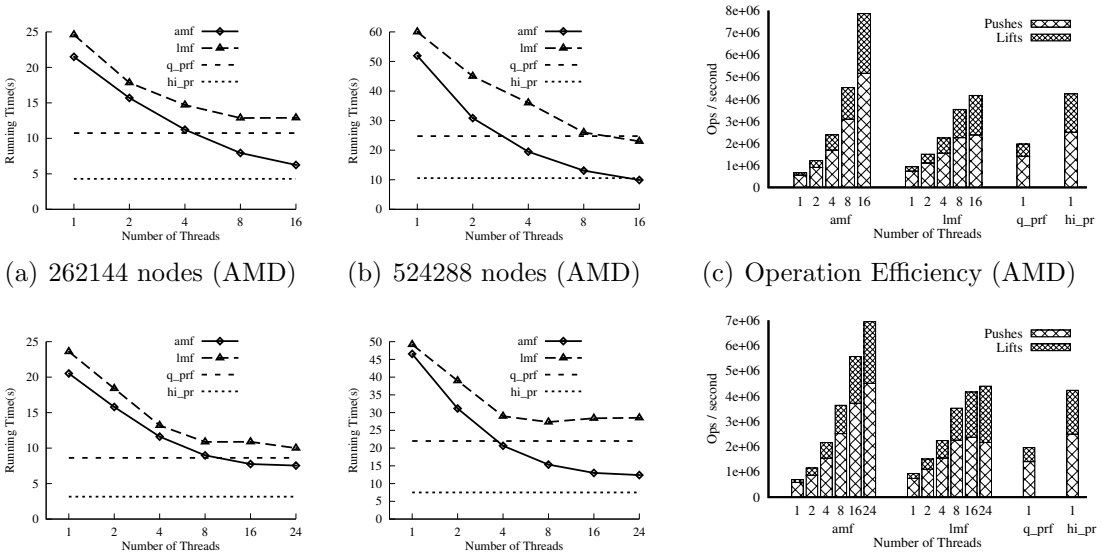
(d) 262144 nodes (Intel) (e) 524288 nodes (Intel) (f) Operation Efficiency (Intel)

Figure 3: Experimental results on Genrmf-long graphs.

for the same vertices so we did not observe the increase of execution for **lmf**. But we expect to observe similar behavior when we further increase the number of threads (as future processors may support). This demonstrated that by avoiding locks, our **amf** algorithm demands less system resources and is therefore able to support more threads on a given hardware platform.

Figures 3 and 4 show the results for the Genrmf-long and Genrmf-wide graphs. Our **amf** algorithm scaled well and outperformed **q_prf** when more than 4 threads were used. When the number of threads reached 16, **amf** even outperformed **hi_pr** which has a lower complexity bound, especially when $|V| \ll \sqrt{|E|}$ for the sparse Genrmf graphs. For Genrmf-long graphs, **amf** achieved an operation efficiency of 11916710 ops/s, which is 1.66 times higher than **hi_pr**'s 4475194 ops/s. For Genrmf-wide graphs, **amf** achieved an operation efficiency of 7859444 ops/s while **hi_pr** only achieved 4230441 ops/s.

In Figures 3 and 4, we can also observe that our **amf** algorithm scales well while the lock-based **lmf** algorithm saw increased execution time when the number of threads



(a) 262144 nodes (AMD) (b) 524288 nodes (AMD) (c) Operation Efficiency (AMD)

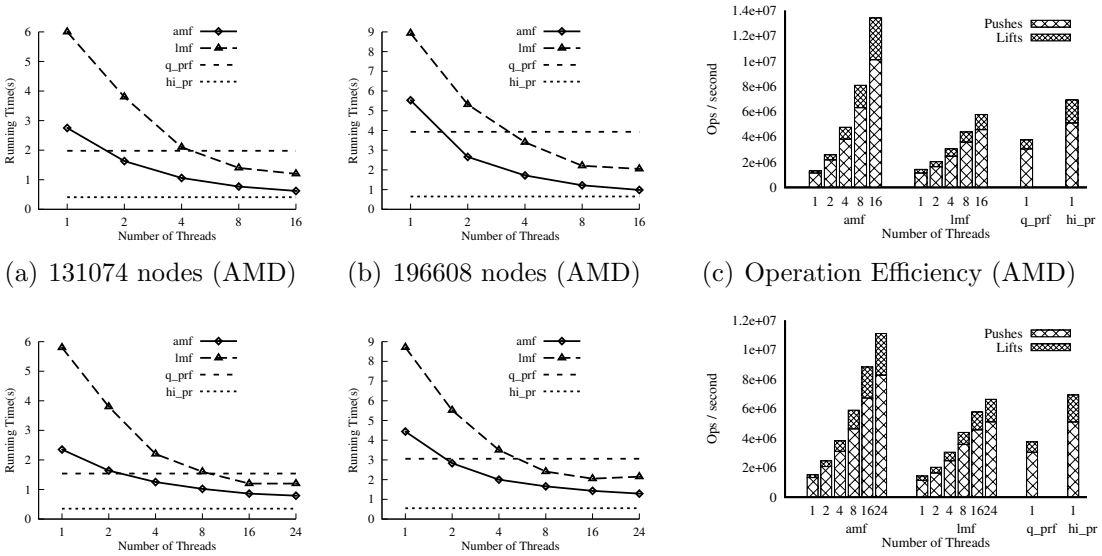
(d) 262144 nodes (Intel) (e) 524288 nodes (Intel) (f) Operation Efficiency (Intel)

Figure 4: Experimental results on Genrmf-wide graphs.

exceeded 16. The **lmf** algorithm also exhibited significantly lower operation efficiency (up to 59% lower than our **amf**). These results demonstrated the effectiveness of avoiding lock usages in our algorithm design.

On the Washington-RLG graphs, **amf** demonstrated the similar scalability and absolute speedup as on the Genrmf graphs (Figure 5 and 6. When we had more than 16 threads, **amf** was 3 times faster than **q_prf** and even approached the execution time of **hi_pr** which holds a lower complexity bound.

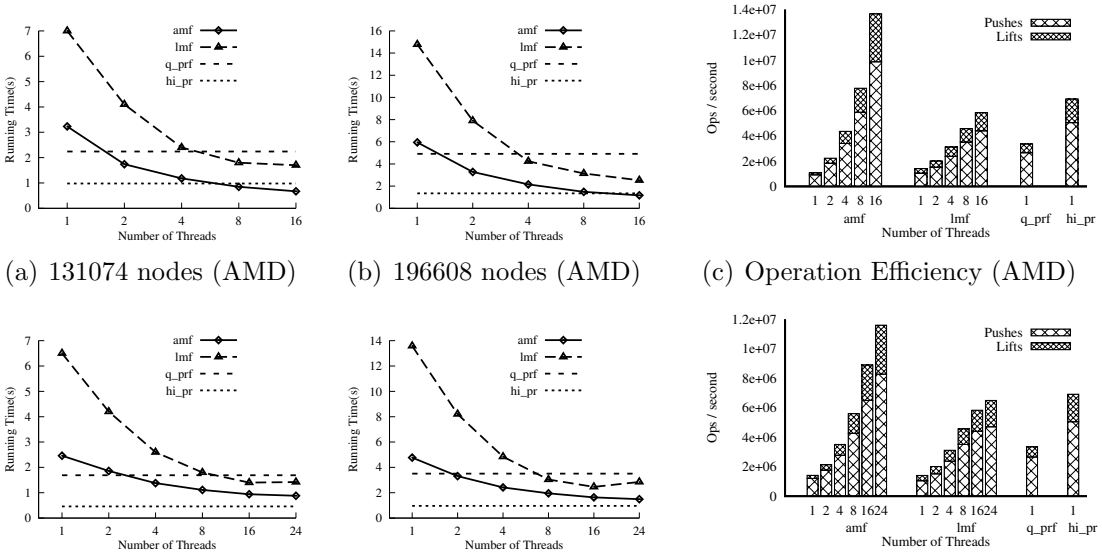
In summary, our experiments show that our **amf** algorithm is a competitive parallel algorithm for the max-flow problem. Unlike the lock-based **lmf** algorithm that often failed to scale when the number of threads exceeded certain threshold, our **amf** algorithm scales well as the number of threads increases up to the maximum number supported by the test platforms. Our algorithm also demonstrated absolute speed up over the well-known sequential push-relabel algorithms, which have lower complexity bound ($O(|V|^3$ or $O(|V|^2\sqrt{|E|})$ than our algorithm ($O(|V|^2|E|)$.



(a) 131074 nodes (AMD) (b) 196608 nodes (AMD) (c) Operation Efficiency (AMD)

(d) 131074 nodes (Intel) (e) 196608 nodes (Intel) (f) Operation Efficiency (Intel)

Figure 5: Experimental results on Washington-RLG-long graphs.



(a) 131074 nodes (AMD) (b) 196608 nodes (AMD) (c) Operation Efficiency (AMD)

(d) 131074 nodes (Intel) (e) 196608 nodes (Intel) (f) Operation Efficiency (Intel)

Figure 6: Experimental results on Washington-RLG-wide graphs.

2.8 Summary

In this Chapter, we presented an $O(|V|^2|E|)$ asynchronous multi-threaded push-relabel algorithm for the maximum network flow problem. The algorithm features

lock-free push and relabel operations. We further developed an asynchronous global relabeling heuristic to speed-up the execution speed of our algorithm. Experimental results demonstrated the effectiveness of the algorithm in terms of both scalability and absolute speed up. This algorithm also suitably fits the emerging CPU architecture where locks have expensive overhead, and the detailed implementation and experiments can be found in [22].

To generalize such asynchronous parallel algorithm design methodology to other problems, one would need to carefully examine possible race conditions when the synchronization requirements are relaxed. Next, we need to reason all the possible outcomes of these race conditions, and then think how to correct them without involving much overhead and introducing more synchronizations. Take the push-relabel algorithm for example, along the execution of the relaxed push-relabel algorithm, there are race conditions because of the arbitrary interleaving of multiple threads. However, the results of race conditions can be reduced to several simple scenarios after we adopt atomic operations, and the new algorithm will correct those violations automatically. In this special case, the correcting operation is a push operation so that it can be seamlessly added in by modifying the original push operation (push to the lowest neighbor instead). This modification doesn't need to add in new operations and also doesn't require any additional synchronization. However, our experience shows it is difficult to generalize this idea to other problems.

Other than relaxing synchronization requirements of existing algorithms, we could also redesign a brand new asynchronous algorithm from the scratch. This direction would possibly be non-trivial and require more efforts, but it is still worthwhile for exploring.

Furthermore, we can also consider another way to improve the concurrency of multi-threads algorithm: improve the programming model. This direction would have more audience and benefit more algorithms. Transactional memory is such an

example. The idea is to protect each critical section by a transaction, and it will optimistically allow multiple transactions to execute at the same time. Only when these transactions cause conflicts, transactional memory will rollback some of the transactions and allow only one to commit. Besides, transactional memory will prevent deadlocks by the system automatically. Therefore, this method can substantially improve the programmability and performance of the multi-threaded algorithms that can be written in "transaction" form.

CHAPTER III

PERFORMANCE MODELING OF TRANSACTIONAL MEMORY

Transactional memory (TM) is a concurrency control mechanism for parallel computing. It provides better programmability than locks [1]. With TM, programmers only need to locally consider the shared-data access and mark the code accordingly, and the underlying TM system will ensure the correctness of concurrent executions. Compared with lock-based schemes, TM is expected to significantly reduce the difficulties of parallel programming and debugging, the vulnerability to failures and faults, and the likelihood of deadlocks.

Multiple factors affect the performance of TM based programs. When a transaction aborts because of a conflict, the computation that has been performed so far will be wasted. If the aborted transaction restarts, it may be aborted again, resulting in further waste. Intuitively, a longer transaction is likely to encounter more conflicts. The implementation overheads will inevitably prolong the length of transactions. Implementation overheads is actually closely associated with the 'optimistic' nature of TM systems: the immediate results *must* be buffered in either software or hardware so that a transaction can be rolled back in the case of a conflict [3]. Buffering takes time and resources, which may increase the contention level and thus intensify conflicts. The research community has been aware of the importance of both factors, which can be witnessed by a large amount of research efforts that are dedicated to explore TM design schemes (e.g. [23, 24]).

The objective of this Chapter is to provide a theoretical model that can reveal the relations between the performance and various key parameters of TM systems,

including the length of transactions, the transaction arrival frequency, the number of conflict detection points (the time points a transaction validate its read/write set), and the computing cost of transactions. In our model, we analyze the run-time behaviors of transactions for a given computer system, estimate the extra time that will be wasted due to conflicts, and obtain the expected execution time needed by the transactions. This approach differs from most existing studies where the focus is on the design and implementation of TMs, and the performance of the TM design is evaluated through simulations or actual executions by using benchmark suites such as STAMP [25]. Such empirical evaluation methods do provide very useful insight to TM studies, but are often unable to isolate the impact of an individual design option (because a typical TM system often integrates a collection of design options). We believe our analytical study would provide a useful tool for TM researches, especially for understanding the complex run-time behavior of transactional execution.

Our analytical model is based on Queuing Theory. In this model, each transaction is a client and the computing system acts as the server responding to the clients. Different from our previous results [26, 27] that focused on uniform transactions, this Chapter studies a general scenario in which multiple types of transactions may have partial conflicts. The correctness of our model is validated through extensive experiments using the STAMP benchmarks [25].

Section 3.1 summarizes various existing TM systems and the related works. The target TM systems of the model is introduced in Section 3.2, and an analysis of the TM systems is in Section 3.3. Section 3.4 presents the queuing model. In Section 3.5, we validate the model by comparing its theoretical performance prediction with the experiment data from actual runs of TM systems. Section 3.6 summarizes the Chapter.

3.1 Background and Related Work

The concept of transactions was borrowed from database systems [28] to enforce the atomicity and isolation for shared memory programming. In TM systems, a transaction completes modifications to shared memory regardless of other transactions. Reads and writes inside transactions should logically occur at a single instant. No intermediate states can be observed or interfered by peers. Every transaction must record its read and write operations in a log, either in hardware or software, until it successfully commits. Upon the detection of a conflict, all the previous memory-access operations of this transaction are rolled back according to its read/write log.

The idea of TM originated in [29] and is formalized by Herlihy et al. [1]. The first STM is proposed by Shavit et al. [30]. In the past decade, numerous STM and HTM systems have been proposed [2, 31–40]. However, researchers found STM and HTM both have inherent shortcomings. Therefore, HyTM has recently been proposed and become the focus of intensive research [41–45].

Various designs have been explored for TM systems. The three key aspects for TM designs are (1) conflict detection, (2) version management, and (3) conflict resolution [24]. **Conflict detection** decides when to detect conflicts, and the two popular design choices are *eager* and *lazy*. The eager option (e.g., in TinySTM [34] and Eazy-HTM [46]) attempts to detect conflict for every memory access. The lazy option (e.g., in TL2 [2] and TCC [32]) may delay the detection to the commit phase, which has been demonstrated to be able to avoid certain conflicts [24]. **Version management** handles the storage policy for permanent and transient data copies. Similarly, the policy can be either *eager* or *lazy*. In the TM systems with eager version management (e.g., TinySTM-WT and LogTM [35]), new data will take place of the old data in the memory and the old data will be logged. In the TM systems with lazy version management (e.g., TinySTM-WB, TL2 and VTM [37]), on the contrary, the old data is kept in place while the new data is logged. **Conflict resolution** manages the actions

to be taken when a transaction encounters a conflict. Many options are available, such as wait, abort self, or abort others.

3.1.1 Summary of TM Systems

In the next , the histories, methods and underlying reasonings of STM, HTM and HyTM systems are described in detail, respectively.

3.1.1.1 HTM Systems

Herlihy and Moss [1] proposed the first TM system. Initially, TM is proposed in the form of HTM which utilizes the existing cache and cache coherence protocol to enforce atomicity of transactions. With only simple additions to the existing hardware, Herlihy's HTM can support atomic transactions that are short enough to complete without context switching. Thus, later HTM systems mostly target on how to support unbounded transactions and enable them to survive context switches.

Hammond et al. [32] proposed transactional coherence and consistency (TCC) which is considered as another HTM system, though it fundamentally change the definition of memory consistency. TCC advocates that memory operations in conventional systems should be replaced by transactions. All the intermediate results of a transaction are buffered in cache and are not broadcast to main memory and all other processors until it commits. Upon receiving these updates, the conflicts can be detected by each processor. When a processor finds the cache is not enough for the current transaction, it simply starts broadcasting updates immediately as they are executed. This processor does not release the bus until the entire transaction completes. Therefore, TCC can support unbounded transactions.

VTM proposed by Rajwar et al. [37] breaks the limitation of on-chip resources. VTM stores transactional state information in the virtual address space of the executing thread, and thus supports unbounded transactions that are able to survive context switches.

Ananian et al. [40] proposed LTM and UTM in the same paper. UTM supports context switches for the transactions, and the size of transactions is only limited by the amount of virtual memory. However, although UTM can provide almost ideal support for transactions, it requires very complicated modifications to the existing hardware. Therefore, LTM is proposed as a replacement, which requires much less hardware modifications. Because LTM still escapes the limitations of on-chip resources, it supports large transactions (only limited by the size of the physical memory), and the transactions can survive context switches.

A more practical system Log-TM is proposed by Moore et al. [35] which is implemented in Simics [47] simulator in conjunction with GEMS [48]. Similarly, Log-TM uses cache coherence protocol to detect conflicts. However, unlike LTM and TCC which buffer all the intermediate results until the commit time, Log-TM writes the new value in-place and log the old values in the main memory. The aborts are handled by software libraries which will walk through the log of old values to restore all memory modifications. The above design causes the transactions to commit very fast but to be aborted considerably slow. However, the authors argue that commits are far more than aborts as they observed in their experiments.

Yen et al. [49] further improved Log-TM by decoupling caches from HTM systems. Traditionally, a HTM system has to add read and write bits to each cache line for the logging. Yen et al. replace these bits with a signature register located inside the processors. They argue that this replacement can save hardware resources and provide convenience for OS to virtualize their system.

Tomic et al. [46] recently proposed EazyHTM combining eager conflict detection and lazy conflict resolution. The cache coherence protocol is modified to make processors tolerate the conflicts until the commit time. A racers-list and a killers-list are added to each processor to record the conflicting relations. When a transaction commits, EazyHTM will check these two lists to figure out which conflicting transaction should be aborted. The authors state that the extra hardware additions can be compensated by the gained performance.

3.1.1.2 STM Systems

The first STM was proposed by Shavit et al. [30]. In their system, a transaction makes updates to a concurrent object only after it broadcasts its updates and declares the ownership of the object. If the declaring transaction fails to acquire the ownership, it aborts and releases the ownerships of its acquired objects. Their system requires all the input and output of a transaction to be known in advance. This requirement limits its application.

A dynamic STM (DSTM) was proposed by Herlihy et al. [50]. DSTM system accesses memory at an object granularity so that it allows transactions to touch a dynamic set of memory locations. Because this technique requires the concurrent objects to be copied at the access time, its performance is poor for large objects. The design of DSTM has been included in Java libraries as DSTM2 [36].

Harris et al. [51] proposed a word-based STM system that uses a hash table for storing ownership records. Their system effectively makes a speculative copy of each word in the transaction and operates on that copy during the transaction. Because their scheme limits the data to be accessed at a word granularity, the problem of large objects are avoided.

Ennals et al. [52] points out that on modern multi-processor machines, cache behavior has a significant effect on performance. His work aims to minimize cache

contention by making some deviations from previous STM designs. He also observed that non-blocking transactions are unnecessary for practical uses and demonstrated the superiority of lock-based STM through experiments. In this scheme, a transaction always attempts to acquire the exclusive ownership (via locks) before it reads a variable, which is similar to that in McRT-STM [53]. All these designs are called encounter-time-locking (ETL) and are closely related to eager conflict detection.

On the contrary, TL2 proposed by Dice et al. [2] uses a commit-time-locking (CTL) strategy. TL2 then evolves to TL2C [54] with the adoption of a distributed clock. In both TL2 and TL2C, locks are acquired only during the commit time. Read/write operations in a transaction are recorded in a read/write log. During the commit time, all read/write locations are locked, and the version numbers of these locations are checked to determine whether the transaction should commit or abort.

Recently, ETL is revisited by Felber [34] et al.. In their proposed TinySTM system, a lazy version management (write-back) strategy is introduced where the memory location is locked at the encounter time, but the updates are stored in a write log and validated only during the commit time. This scheme combines lazy version management with eager conflict detection to lower the cost of aborts.

RSTM proposed by Marathe et al. [23] is another object-based STM system. Different from all other STM systems, it equips multiple types of contention managers [55, 56]. Experiments show that different contention managers possibly lead to totally different performance for the same workload.

SwissTM recently proposed by Dragojević et al. [33] contains both object- and word-based implementations. SwissTM adopts two different contention-management policies in both implementations, as the authors state that a static contention manager cannot suit all types of transactions. Moreover, SwissTM also differentiates write-after-write (WAW) conflicts from read-after-write (RAW) conflicts. WAW conflicts are detected eagerly, and RAW conflicts are detected lazily.

3.1.1.3 HyTM Systems

The idea of HyTM was proposed by Kumar et al. [42] and Damron et al. [43] almost at the same time. However, their proposed designs are not identical. Kumar et al. started from the STM side (DSTM) and proposed to add an addition transactional buffer recording the intermediate results to accelerate the logging operation of STM. However, the hardware additions in this design seems to be too expensive than the commercial processors can afford, and their HyTM system has to abort the transaction on a context switch. On the contrary, Damron et al. built their HyTM system from the HTM side while assuming they already had a bounded and best-effort HTM system. They implemented a set of novel data structures to connect their STM and HTM subsystems. In their scheme, if a transaction finds its size exceeds the limits of hardware, it will be aborted and restarted in the software. Therefore, transactions committed in hardware will be considerably faster than that in software. This design also causes difficulties in incorporating contention manager with hardware transactions.

Saha et al. [44] proposed to extend the instruction-set architecture (ISA) to provide architectural support for STM systems. Six new instructions are added to help programs mark or unmark a certain memory location. In their design, transactions are always executed in software, and those special instructions are used to shift the loads of the logging and conflict detection to the hardware. This design requires only trivial additions to the existing processors but achieves better performance than the original STM system.

RTM proposed by Shriraman et al. [45] is another HyTM design. RTM is based on alert-on-update [57], a novel architecture for shared memory programming. To support unboundedness, RTM restarts a transaction in a more conservative and slower "overflow mode" when the execution time of the transaction exceeds a single quantum, or when the size of the transaction exceeds the capacity of the cache.

Baugh et al. [41] recently proposed another HyTM system. In their design, software and hardware transactions can be executed concurrently, and their contention manager can switch hardware transactions to software upon the detection of conflicts. However, they still did not solve the problem of how to manage contentions across hardware and software.

The performance of a TM system is affected by multiple types of overhead. The *abort* and *logging* costs of a transaction are two major types of overhead. These two types of costs actually are caused by the "optimistic" nature of TM. Because all transactions are issued out optimistically, some of them may conflict with each other. To keep the shared data consistent, only one of the conflicting transactions can commit and all the others should be aborted. The computations that performed by the aborted transactions must be abandoned, which is a huge cost. Moreover, to provide a transaction the capability of rolling back its previous memory operations upon an abort, all the intermediate results of the transaction must be logged in either software or hardware. Fortunately, the number of conflicts and aborts can be reduced by employing a more sophisticated contention-management (CM) policy [55], and the logging can be accelerated by adopting hardware mechanisms.

Different TM implementations lead to different amounts of costs for the abort and logging. Hardware transactional memory (HTM) uses caches or special buffers to log the intermediate results so that it causes very little logging overhead. However, HTM utilizes cache coherence protocol to detect conflicts. Thus, the overhead caused by aborts in a HTM system may be huge because of the simple conflict detection and resolution strategies (e.g., livelock in extreme cases [24]). In addition, HTM also suffers from various limitations: it is expensive or even unable to support large transactions, difficult to survive context switches, and inflexible to be adjusted by the programmers (e.g., the granularity of a HTM system is fixed to the size of the cache line). STM is more flexible than HTM and does not have the above drawbacks, but its

performance is hurt by the large logging overhead. Thus, hybrid transactional memory (HyTM) [42,43] and hardware assisted software transactional memory (HASTM[1]) [44, 45] have recently been proposed. HyTM integrates both hardware and software in the design so that it can provide STM-like flexibility and HTM-like performance.

3.1.2 Performance Modeling of TM

Most of the previous studies evaluate the performance of TM systems through experiments based on either simulations or actual executions. Such empirical evaluation methods have provided very useful insights to TM studies. On the contrary, there have been very few theoretical studies on the performance of STMs. Our previous analytical models ([26,27]) adopted continuous time Markov chain (CTMC) and studied the mean transaction execution time. These models focused on the scenario of uniform transactions. The model developed by Heindl et al. [58] adopted discrete time Markov chain (DTMC) to investigate the impact of transaction conflicts. A "tagged" transaction is analyzed to represent the overall performance of TM systems. However, this model studies the expected number of retries before a transaction commits, which is not a direct measure of execution speed (because this model does not study the time needed to complete a retry, which may vary depending on the number of active transactions). Porter et al [59] developed a tool called Syncchar which models the workload performance of TM. This model statically estimates D_n, the expected number of pair-wise conflicts assuming all n transactions execute simultaneously, and assumes that transactional execution of n threads will be slowed down by D_n times. This model, however, does not take into consideration that conflicts are dynamic and transactions with conflicting read/write-sets may not be executed simultaneously.

Compared with the existing analytical works, we study the dynamic run-time behaviors of transactions in this Chapter. Our analysis supports multiple types of

[1]HASTM is considered as a special type of HyTM in the following discussion.

transactions, including partially conflicting transactions.

3.2 Target TM systems

In this section, the characteristics of the target TM systems are specified, and the notations for the following discussion are defined.

Let N denote the number of threads in the system. And we assume each thread will be executed by one processor. Each thread is capable of executing each of the m transaction types. We assume that when a thread is executing a transaction, it does not start another one. The scenarios of nested transactions are reserved for the future study. When a thread is not executing any transactions, we call it a *potential thread* because it can potentially start a new transaction.

$X_1, X_2, \cdots, X_m$ denote the m types of transactions, where the proportion of X_i over all the instances of all types of transactions is p_i. X_i has k_i conflict detection points. Different TM designs will invoke different numbers of conflict detection points for the same transaction. Each shared read/write has a chance to be a conflict detection point. However, because of the overhead, a transaction will not detect conflicts on every read or write. When no conflicts occurs (e.g only one thread is running), the probability that a thread start a transaction within Δt time is $\lambda \Delta t + o(\Delta t)$; and the probability that a thread reaches a conflict detection point with Δt time is $k_i \mu \Delta t + o(\Delta t)$. λ and μ represent the arrival and service rates respectively on the condition that the transactions are executed sequentially. When multiple threads are concurrently executing transactions, λ and μ may decrease if the shared computing resources (e.g., shared cache and interconnect) are saturated. We use C to denote the maximum rate at which the underlying computer system can supply the shared computing resources, and transaction X_i consumes these resources at a rate of c_i. We assume the shared computing resources will be evenly distributed among threads. Therefore, the actual λ and μ need to be corrected accordingly (details are

in [60]).

When two transactions conflict, they must be accessing the same data, and more importantly, they should be running concurrently. Many complex conflicts exist, including Read-after-Write (RAW), Write-after-Write (WAW), and Write-after-Read (WAR). To balance complexity against accuracy, we model the conflicts with p_{con}, the probability that two transactions conflict (which is detected at conflict detection point as well as commit point). With this simplified conflict model, the eager and lazy conflict detection strategies can be described by changing the number of conflict detection points. For instance, eager conflict detection will encounter more conflict detection points so that it can have a higher probability to detect conflicts earlier. Upon the detection of a conflict, a transaction will abort itself. This strategy is adopted by many TM systems (e.g., TinySTM with suicide option). In the model, when two transactions conflict, we assume the conflict will be detected by the transaction that has made less progress towards completion. This assumption is statistically reasonable: an *older* transaction is expected to have accessed more shared variables than a *younger* transaction, so the younger transaction has a higher probability than the older one to detect a conflict.

Figure 7 shows an example of the target TM systems. X_a, X_b and X_c are three different transactions issued by two threads. X_a illustrates that each transaction have a start, a commit and several (k_a for X_a) conflict detection points. Transaction X_a finishes with rate μ, and we assume each conflict detection point will be finished at rate $k_a\mu$. When executing transaction X_a, Thread 1 cannot issue new transactions. However, after finishing X_a, Thread 1 issues a new transaction X_b with rate λ. If at the same time, Thread 2 issues a transaction X_c, it will detect the conflict with X_b and abort X_c. X_c will keep retrying until it commits.

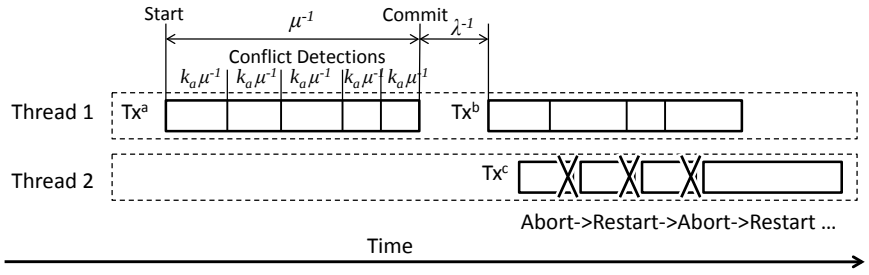
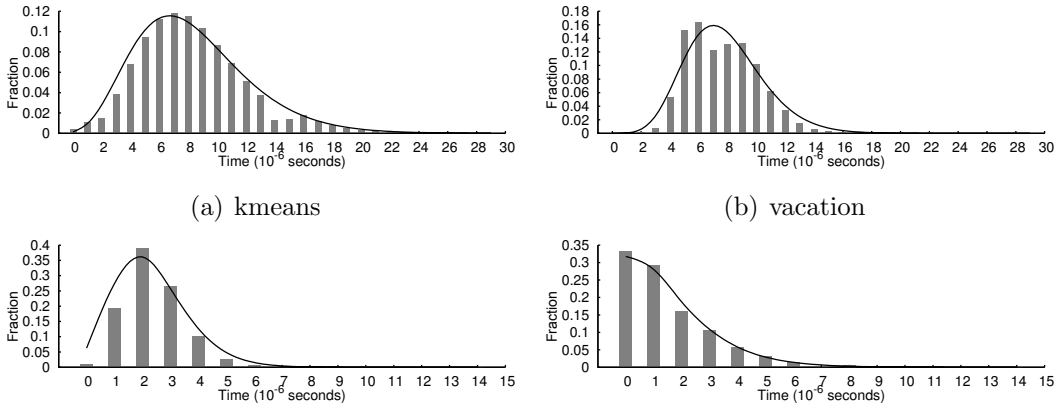


Figure 7: Illustration of transactional execution of the target TM systems.

3.3 Analysis of the TM systems

3.3.1 Statistical Characteristics of Transactions

From the profiling results of the STAMP benchmarks [25], Erlang distributions were observed for the transaction execution time. In the profiling, the benchmarks were configured with recommended parameters in [25] and executed by eight threads. The results of four representative benchmark programs are shown in Figure 8. The Erlang distribution has two parameters: the shape k, which is a non-negative integer, and the rate μ, which is a non-negative real number. When k equals to 1, the Erlang distribution become the exponential distribution as shown in Figure 8 (d).



(a) kmeans (b) vacation

(c) ssca2 (d) intruder

Figure 8: Erlang distributions of transaction execution time (TinySTM, eight threads, Redhat 5.4 x86_64, gcc 4.1.2. The bars form the histogram of the execution time, and the solid lines is the fitted curve).

The above observation can be explained as follows: a transaction needs to pass

several important time points during its execution (e.g., the start point, the intermediate conflict detection points, and the commit point). We assume the probability p of hitting such a time point is proportional to the amount of elapsed time Δt, i.e. $p = \mu\Delta t$ for some constant μ. It can be easily shown that the time between two time points obeys exponential distribution with parameter μ. When a transaction has k such segments, the sum of the k exponentially distributed segments is then equal to an Erlang distribution with parameters k and μ. The linear relation between p and Δt reflects the behavior of a typical program: given a longer period of time, a program is expected to make more progress.

Therefore, the execution of a transaction X_i is modeled as a sequence of events: a `Start`, $k_i - 1$ `Conflict Detection`s, and a `Commit` (for notational convenience, the commit point is considered as the k_i^{th} conflict detection point), where the time between two consecutive events obeys the exponential distribution. When a transaction is aborted, it needs to repeat the whole sequence of the events over again until it can successfully commit.

3.3.2 Impact of Transactional Congestion

In practice, a computer system has finite computing resources. As defined in Section 3.2, the overall shared computing resources has a maximum value C When multiple threads are running concurrently in a system, the demanded amount of computing resources may exceed the system capacity and thus cause congestion. We assume every thread will be treated fairly in sharing the computing resources. We use R to denote the ratio between demands and supplies, which can be calculated as

$$R = \frac{n \sum_{i=1}^{m} p_i c_i}{C} \tag{1}$$

where n is the current number of threads. When $R > 1$, the computing system will be overloaded and all the threads will be slowed down. We calculate the threshold of

n by

$$n_{th} = \frac{C}{\sum_{i=1}^{m} p_i c_i} \tag{2}$$

In our model, we assume when the system is congested, namely $n > n_{th}$, all the processors will be affected equally. Their running speed will be decreased by a factor of R. Therefore, the arrival and service rates of transactions (λ and μ) running on these processors will also be decreased by a factor of R:

$$\lambda_{cor} = \begin{cases} \lambda & n \leq n_{th} \\ \lambda/R & \text{otherwise} \end{cases} \tag{3}$$

$$\mu_{cor} = \begin{cases} \mu & n \leq n_{th} \\ \mu/R & \text{otherwise} \end{cases} \tag{4}$$

and we use these corrected λ_{cor} and μ_{cor} in the following analysis.

3.4 Queuing Model of TM systems

With the above notational preparation and initial analysis on the execution of transactions, a TM system can be modeled as a finite source queuing system where each transaction is a client and the computing system acts as the server responding to the clients. More importantly, the client in the system may quit a current service and repeat the service process again (transactions may be aborted at a conflict detection point and restarted from the beginning). The states of the TM systems are shown in Figure 9. N is the number of threads, and $K = max(k_1, k_2, ..., k_m)$ represents the maximum number of conflict detection points that transactions may have in the target TM system. k_i denotes the number of conflict detection points for transaction X_i. The states are defined as follows:

- **State** [0] denotes the state of the TM system that no thread is executing any transaction.

- **State** $[n, x]$ denotes the states of the system where (1) n transactions are running; (2) at least one transaction will commit; and (3) the first transaction that will commit is working towards its x^{th} conflict detection point. The transaction that will commit first is named as the *working transaction*, and the thread executing it as the *working thread*.

- **State** $[n']$ denotes the states of the system where (1) n transactions are running; and (2) all running transactions will be aborted and restarted at the next conflict detection point (or commit point). The aborted transaction will be restarted immediately after the abort. The transaction that can commit earlier than others will become the new working transaction, which is denoted by *quasi-working transaction*, and the thread executing it is denoted by *quasi-working thread*. Note that the quasi-working thread is not necessarily the one that is first aborted.

The possible transition routes among the states are also presented in Figure 9. Each of these routes is based on a possible system transition invoked by the arrival of an event. As defined in Section 3.3, there are three possible types of events: `Start`, `Conflict Detection` and `Commit`. Only the events issued by the working thread, quasi-working thread or potential thread are analyzed, because other events will not cause state transitions. Except for `Start`, each event has two possible consequences for the current system state, "Succeed" and "Abort". "Succeed" means the transaction issuing this event will continue to the next conflict detection point. "Abort" denotes that the arrival event will cause the system to abandon the previous work and start rolling back. `Start` event does not have the "Abort" consequence because, as we assumed in Section 3.2, if a thread fails to start, it will restart the transaction

immediately.

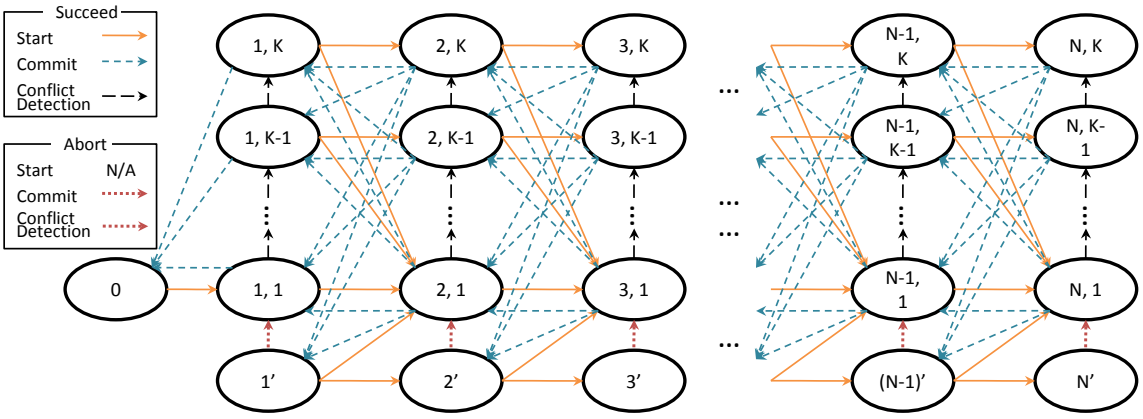


Figure 9: State Transition Diagram of TM Queuing Model based on CTMC.

In particular, the transitions invoked by `Start` and `Commit` events have multiple cases. When a thread issues a new transaction at the state $[n, x]$, the number of threads in the system will be incremented by one, from n to $n+1$. However, depending on whether the new thread replaces the current working thread or not (if the new transaction can commit early than the current one), the state will transit to either $[n, 1]$ or $[n, x]$. For a `Commit` event that succeeds, the transition also has multiple destinations. When the working transaction commits, the number of threads in the system is decremented by one. The destination depends on the status of the next working transaction. If no working transaction exists, the destination state is $[(n-1)']$; if the next working transaction is approaching its y^{th} conflict detection point, the destination state is $[(n-1), y]$.

Furthermore, not all the states can see all three types of events. For instance, when the system is at $[0]$, the only possible coming event is a `Start`, because no transaction exists in the system at that moment. Similarly, states $[n, K]$ have no `Conflict Detection` events and states $[N, x]$ have no `Start` events.

After identifying the possible transition routes, the transition rates for each route can be calculated. These rates are derived by enumerating every possible case that

Table 1: Model input parameters

	genome	**intruder**	**kmeans-high**	**kmeans-low**	**labyrinth**	**ssca2**	**vacation-high**	**vacation-low**	**yada**
Time in Tx	97%	43%	33%	30%	96%	25%	86 %	86%	97%
λ/μ	32.3	0.75	0.5	0.4	24	0.33	6.1	6.1	32.3
Contention	Low	High	High	High	Low	High	Low	Low	High
p_{con}	0.10	0.50	0.90	0.75	0.52	0.43	0.26	0.25	0.70
Consumption	Low	High	Normal	Normal	High	High	High	High	High
$\bar{c_i}$	0.9	4	1	1	2	2.1	1.2	1.2	2.2
Tx Types (m)	5	3	3	3	3	3	3	3	6
Percentage of Tx (p_i)	(0.01, 0.50, 0.41, 0.07, 0.01)	(0.34, 0.33, 0.33)	(0.75, 0.01, 0.24)	(0.75, 0.01, 0.24)	(0.5, 0.001, 0.499)	(0.005, 0.99, 0.005)	(0.98, 0.01, 0.01)	(0.90, 0.05, 0.05)	(0.01, 0.17, 0.24, 0.17, 0.24, 0.17)

* **bayes** benchmark is excluded because of its non-deterministic finishing conditions as noted in [25], which makes the comparison against the deterministic result generated by theoretical model less meaningful.

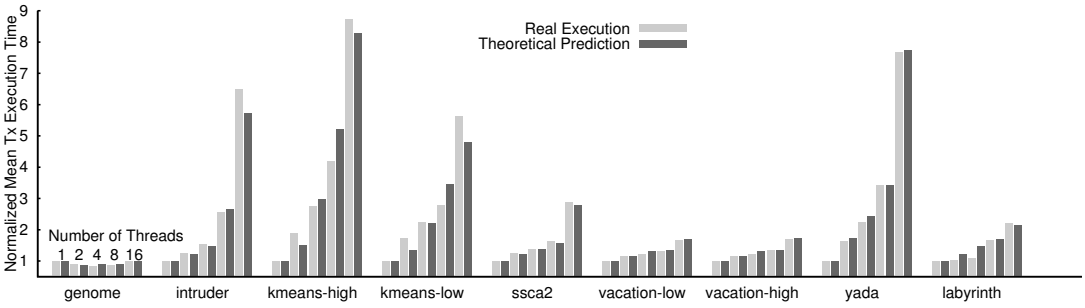


Figure 10: Comparison between real executions and theoretical prediction for different benchmarks

may happen at each state and then combining the rates along the same routes. A detailed derivation can be found in [60].

3.5 Experiments and Discussions

We validate our model against the STAMP benchmark suite, which is widely accepted by the researchers as the typical TM workload. STAMP has an open infrastructure that allows the use of various TM implementations, and we tested TL2, TinySTM and SwissTM [33]. These are three state-of-the-art TM implementations that have demonstrated good performance. The STAMP benchmark programs are configured with the suggested configurations in [25]. Unless otherwise indicated, TL2 is configured with the 'lazy' option (featuring lazy conflict detection and resolution), TinySTM

is configured with the 'ETL-WB-Suicide' option (featuring encounter-time-locking, write-back, and suicide upon detection of conflict), and SwissTM by itself is configured with a mixed conflict detection strategy (eager WAW and lazy RAW conflict detection, transactions having less work will be aborted) [2]. The experiment platform had 24 cores (4 Intel Six-core Xeon E7450 2.4GHz) and 64GB DDR2 800Hz shared memory. The operating system (Redhat Enterprise 5 distribution 64-bit) ran Linux kernel version 2.6.18 and gcc version 4.1.2 was used to generate the executables.

We first examine the accuracy of our model when modeling different applications. TinySTM is selected as the TM system. During the experiments, we first profile the STAMP benchmarks with 1 thread, then use our model to predict their performance of 2, 4, 8 and 16 threads. The prediction results are compared against the real execution results. STAMP contains a list of benchmarks with different characteristics as listed in [25]. The characteristics include the percentage of transactional execution time, transaction length, and average retries per transaction (contention level). The characteristics in [25] were obtained using simulations based on TL2, which deviate slightly from our results on real machines using TinySTM. In particular, we used hardware Time Stamp Counter (TSC) on x86 architectures as the high resolution timer. Although this is the lowest overhead timer that we are aware of, time measurement inevitably introduces certain instrumentation overheads for the TM system. The impact of such overhead is negligible for most benchmark programs as their transactions consist of thousands of instructions, much longer than the instrumentation. But for benchmark programs with very short transactions (*kmeans* and *ssca2*), the impact is noticeable. The transactions were prolonged by the instrumentation. In our analysis, we treat such instrumentation overheads as part of the overheads associated with the TM implementations.

[2] Although TL2 and SwissTM adopt different conflict resolution strategies other than "suicide", the prediction error of our model still relatively low. So we include their results as well.

The parameters describing the properties of the benchmark programs are summarized in Table 1 and were used as inputs to our model. The number of transaction types m was obtained by examining the source code. We also measured p_i for each type of transactions by calculating the number of tries they had to make before the commit. We fixed the service rate $\mu = 1$ in our calculations as we use the normalized transaction execution time as the performance metric. According to the definition in Section 3.2, we can calculate the actual λ as follows:

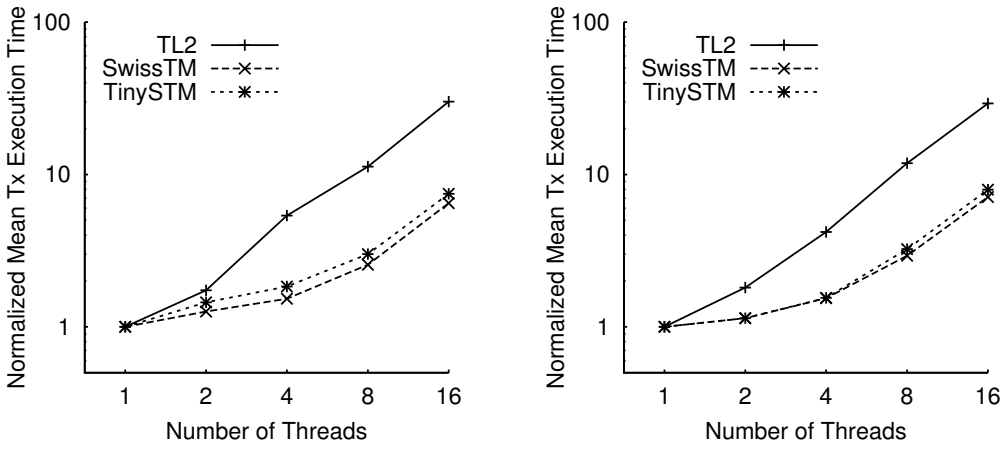
$$\lambda = \frac{T\mu}{1 - T} \tag{5}$$

where T is the time spent in transactions for each benchmark (Time in Tx in Table 1). The contention level reflects the conflicting probability p_{con}, and it is related to the average retries per transaction that can be observed. The 'consumption' row in Table 1 depicts the demands on the resources. It is estimated by checking the ratio of the number of read/write barriers over the transaction length, all of which can be found in [25]. When we set the total computing resources C to 24 (assuming the computer can support 24 cores without performance degradation when each of them consumes a maximal of $c_i = 1$ unit of resources), we choose the values of c_i according to its consumptions and the average number of c_i is presented in Table 1, where a values larger than 1 represents a higher computing resources consumption than normal code.

The results of actual execution and our model based prediction are illustrated in Fig. 10. We normalize the mean transaction execution time to that of a single thread. It can be seen that our theoretical prediction fits actual executions very well. The average error rate is 7.9%. The maximum error rate is 14.7% for *kmeans-low* with 16 threads. In both Fig. 10, *kmeans-high* and *genome* have the worst and best scalability respectively, which are mainly determined by their contention levels. Similar results can be found in [4]. Note that we present the mean transaction execution time, while

other profiling results present the total execution time which also includes the time costs of the sequential portions.

Next we verify the capability of our model in modeling different TM systems. Although different conflict resolution strategies are adopted in TL2 and SwissTM, our model still exhibits good prediction accuracy. We focus on the *intruder* benchmark because it has three types of transactions with equal percentages and the length of transactions are roughly the same. This gives us great opportunity to easily tune the number of conflict detection points k_i in our model. We use the same value for k_1, k_2 to k_3 and use k for short in the following discussion. We tested all three TM systems: TL2, TinySTM and SwissTM. Because TL2 is configured with lazy conflict detection and resolution, it will even tolerate WAW conflicts so that it has the smallest number of conflict detection points and least overheads associated with conflict detections. TinySTM is able to detect most conflicts, thus it has the largest number of conflict detection points and also with the highest overheads. The number of conflict detection points and overheads of SwissTM is between TL2 and TinySTM because it has a mixed conflict detection strategy. Therefore, we set k for TL2 to 1, TinySTM to 15, and SwissTM to 10, and $\bar{c}_i$ for TL2 to 1, TinySTM to 4, and SwissTM to 3. Fig. 11 compares actual execution results against our model prediction. The results show that TL2 has an exponentially degrading performance when the number of threads increases. We believe this is primarily due to the lazy option used in TL2 that causes a transaction to waste time in computation that will eventually be aborted. TinySTM and SwissTM have comparable performance on *intruder*, but because TinySTM invokes more conflict detection than SwissTM (more conflict detection points), SwissTM is a little bit faster than TinySTM especially when the number of threads increases. Our model can describe these differences and make the prediction very close to the real performance.



(a) Real Execution (b) Theoretical Prediction

Figure 11: Comparison between real executions and theoretical prediction for different TM implementations

3.6 Summary

In this Chapter, we developed a novel theoretical model to predict the performance of TM systems. Based on the statistical characteristics of transactions, CTMC is employed to model the TM systems: every transaction is a client requesting services and the computer is the server responding these requests. The start, commit and conflict detection events in transactional execution are directly mapped in our model to represent state transitions in the TM systems. By calculating the probability of every state where the TM system may stay, we obtain the mean transaction execution time to evaluate the performance of target TM system. Our model is validated through extensive experiments comparing the results of real execution against our theoretical prediction.

Our model achieved an average error rate of 7.9% in the comparison against real TM systems for STAMP benchmarks. Different from previous performance models, our CTMC model can capture the run-time behaviors of TM systems.

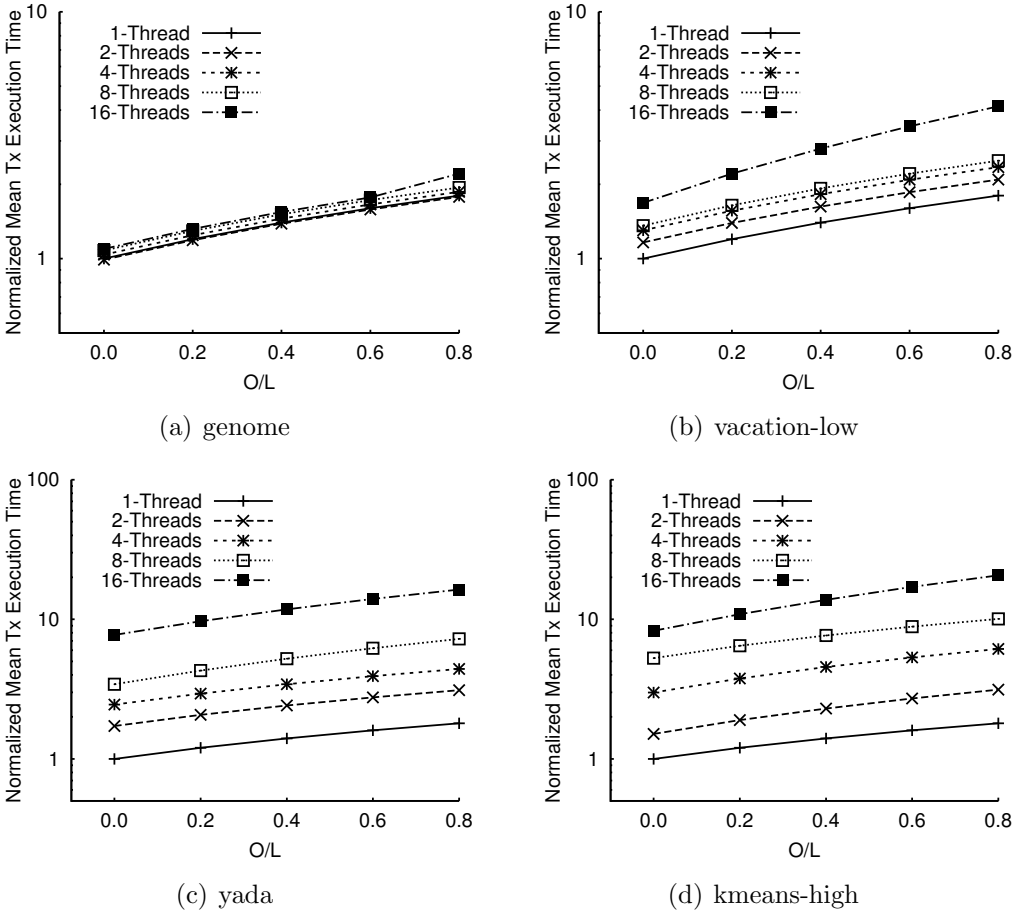
CHAPTER IV

ADAPTIVE CONTENTION MANAGEMENT FOR STM SYSTEMS

Contention manager (CM) is a crucial component of TM systems and has been extensively studied in [55,56,61,62]. CM decides how to resolve a conflict after it is detected (e.g. abort one of the conflicting transaction) and how to avoid it from happening again (e.g. add a random backoff before restarting a new transaction). The goal of the contention manager is to guarantee forward progressing and increase the transaction throughput. At the same time, livelock, deadlock, and starvation are avoided by the contention manager. An efficient contention manager could effectively improve the performance of TM systems, which can be validated by the model presented in Chapter 3.

As discussed in Section 3.3, each transaction is associated with many types of overhead such as logging overhead, rollback overhead, and abort overhead. Four representative benchmark programs from STAMP suite including *genome*, *vacation-low*, *yada* and *kmeans-high* are chosen. The overhead of SwissTM is used to be the baseline. The amount of the overhead O is tested linearly from 0 to $0.8L$ where L represents the original transaction length. The results are summarized in Fig. 12. The average transaction execution time is used to estimate the performance, and the figures are plotted in logarithmic scales. It is shown that with the linear increase of the overhead, the performance deteriorates exponentially on all four benchmarks. On the other hand, it can be seen that the contention level has a noticeable impact on the extent of performance degradation. These four benchmarks contain different workloads so that they cause different level of contentions. From low to high, their contention

levels are in the order of *genome*, *vacation-low*, *yada* and *kmeans-high*. *genome* has very low contention level and resource consumption, so the increase of overhead has little impact on its performance. When the contention level increases, the impact of overhead becomes more and more remarkable. Furthermore, when the number of threads is increased to intensify the contentions, *genome* is obviously more robust than all other three benchmarks. In summary, TM systems with high-contention level will be more sensitive to the overhead. Reducing executing overhead and lowering the contention level (e.g., through a carefully designed contention manager) will be helpful for the performance of TM systems.



(a) genome

(b) vacation-low

(c) yada

(d) kmeans-high

Figure 12: Impact of different amounts of overhead.

Because CM is important to the performance of STM systems, CM has received intensive research attentions and a large variety of schemes have been designed to

explore the trade-offs between performance and run-time overhead [55,56,61,62]. For example, a simple CM may always choose to self-abort a transaction to resolve all the conflicts. Such simple designs have low run-time overhead because the decision is preset. For another example, a complicated CM may favor an older transaction, which aims to preserve existing computing efforts but requires more bookkeeping and higher decision making cost. Unfortunately, as shown in Section 4.2, there does not exist a single CM that performs well for all the transactional workload, and the performance variation of the CMs can be significant.

More importantly, there is no general method to identify a suitable CM scheme even when the workload is known. Given the large variety of proposed CM schemes, a natural solution would be profiling the workload with multiple CMs and then selecting the best one. However, existing STM systems do not support such automatic adaptation and require the programmers to manually perform the profiling and "hard-code" the best choice in the programs. This is against the design objective of TM — TM expects programmers to focus on determining where atomicity is necessary, rather than on the mechanisms that enforce it. The necessity for the manual profiling and selection would make TM less attractive.

In this Chapter, we argue that adaptation is necessary and feasible for the contention management for STM systems. We demonstrate that the performance of CMs is sensitive not only to the type of workload but also to the underlying system platforms. We present an effective profiling method for the adaptation, and use it to develop an adaptive contention manager (ACM) on both TinySTM [34] and RSTM [23]. In our proposed method, we dynamically adjust two key parameters, i.e., the profiling interval and the profiling length of each CM, to reduce the profiling overhead for any type of workload and platforms. We also propose to use logic-time to measure the profiling length. The effectiveness of the proposed ACM schemes is validated through extensive experiments on two platforms (x86 and powerpc). The

main contributions in our Chapter are as follows:

1. We propose a dynamic profiling framework that searches for and applies an optimal CM during the execution of STM workloads.

2. We propose two logic-time based methods to characterize the profiling length of each CM. Particularly, the abort-based method achieves better performance than traditional physical-time-based methods (up to 25%).

The rest of the Chapter is organized as follows: Section 4.1 summarizes the background and related works. Section 4.2 justifies the necessity and feasibility of adaptation. We propose our profiling-based adaptive contention manager in Section 4.3. Section 4.4 presents our implementation details, and Section 4.5 reports the experimental results that validate our new approach. Section 4.6 concludes the Chapter.

4.1 Background and Related Work

In an STM system, conflict resolution is handled by the CM. Three possible decisions may be made by a CM:

1. **Abort-other**: when a transaction detects that it conflicts with another transaction, it will kill the other transaction to ensure the validity of its own copy of the shared data.

2. **Abort-self**: when a transaction detects that it conflicts with another transaction, it will abort itself to ensure the data validity of the conflicting transaction.

3. **Backoff**: two types of backoff schemes exist: (1) when a transaction detects a conflict with another transaction, it stalls itself for a certain period of time, and then re-checks for data validity upon returning from the stall. (2) when a transaction aborts due to a conflict, it backoffs for a period of time before it

restarts. Note that other terminologies may be used to name these schemes. For example, scheme 1 is called "wait" in RSTM.

An ideal CM is expected to (1) minimize the wasted work, (2) avoid future conflicts, and (3) reduce the overhead of executing the CM itself. Because it is often difficult to achieve the three objectives simultaneously, a wide variety of CM schemes have been studied to explore the design trade-offs. We categorize CMs below based on their primary optimization objectives:

1. CMs that emphasize on minimizing the wasted work. These CMs evaluate the conflicting transactions and choose to abort the one that has performed less computation. Some CMs in this category may attempt to backoff before aborting a transaction. Example CMs include:

   - `Timestamp`: always aborts the newer transaction. The start time can be read from the system clock (`Timestamp` in RSTM) or a globally maintained counter (`Greedy` in RSTM and `Timestamp` in TinySTM).
   - `Karma`: always aborts the less-productive transaction. The productivity of a transaction can be evaluated by the size of its data set (reads and writes). Variations of `Karma` may assign more weight to writes (e.g. `Whpolka` in RSTM) or to transactions that already aborted others (e.g. `Eruption` in RSTM).

2. CMs that emphasize on reducing CM overhead. Such CMs often focus on implementation simplicity and does not perform bookkeeping. Example CMs include:

   - `Aggressive`: always aborts the other transaction.
   - `Suicide`: always aborts self. (It is also called `Timid` in RSTM).

- `Polite`: always exponentially backoffs for a number of times, and eventually aborts the other transaction. (This CM is unavailable in TinySTM).

3. CMs that attempt to reduce future conflicts. These CMs are often derived from the above CMs and applies backoff to the transactions that were recently aborted. For example:

   - `AggressiveD`: always abort the other transaction and ask it to backoff for certain period of time (`AggressiveD` in TinySTM asks a transaction to backoff until the lock that caused the abort is released; A variation `AggressiveR` in RSTM backoffs a fixed amount of time).
   - `SuicideD`: abort self and backoff before a restart.
   - `KarmaD`: `Karma` with backoff.
   - `TimestampD`: `Timestamp` with backoff.

In addition to the static CMs summarized above, dynamic CM policies have also been studied intensively.

Guerraoui et al. [61] proposed a framework called polymorphic contention management that allows CM to be changed on-the-fly. But it did not address how to choose the CMs during run time.

There are also attempts to automatically choose a CM policy from two CM policies, such as SwissTM [33]. In SwissTM, they analyzed the suitable cases for two CM (`Suicide` and `Timestamp`), and based on the experiments, a fixed threshold of transaction size is set to decide which CM to use.

Heber et al. [62] implemented an adaptive algorithm that could automatically switch to serialize transactions when the contention level is high. They demonstrated through experiments that this adaptive method could effectively reduce the abort rate of the STM system. But this work uses the contention level as the only 'cue' for adaptation, and the only adaptation strategy is to fall back to lock-based schemes.

For adaptive contention management, Frank et al. [63] proposed a 'reinforced-learning' scheme on DSTM that uses a separate thread to profile the CMs (i.e. throughput) and poll to choose the best CM during the last execution period. The interval between selection is tuned and fixed to one second in [63]. This scheme can achieve adaptation among CMs by profiling target workload at run-time. However, this strategy does not solve the problem on how to properly choose the length of adaptation interval.

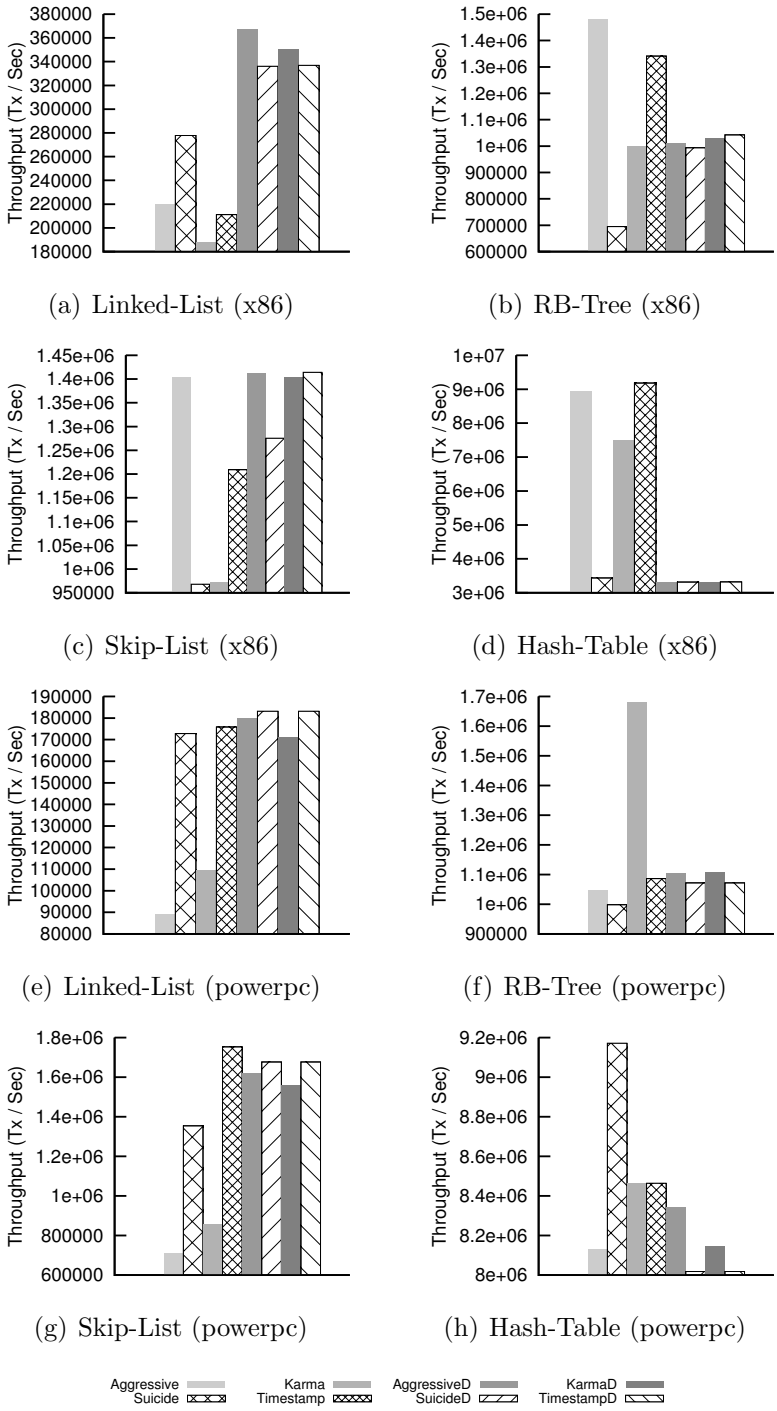
Compared with the above methods, our proposal systematically explores the profiling method for STM performance tuning, and proposes a dynamic adjustment algorithm for profiling interval and length. We also propose to use logic-time to measure the profiling length, and we show this method outperforms the traditional physical-time-based method.

4.2 The Necessity of Adaptive Contention Management

CM has attracted a lot of research attention because of its importance. A large set of CMs have been proposed in the literature and many STM systems are released with multiple choices of CMs. For example, the latest TinySTM 1.0.0 integrates five basic CMs and researchers can easily plug in other more complicated CMs; RSTM release 5 [23] includes a pool of over 20 CMs for programmers to choose from.

Based on different design heuristics, the CMs can be simple (e.g. `Suicide` that always causes a transaction to abort itself in case of conflict) or sophisticated (e.g. `Timestamp` that favors older transactions), where the more sophisticated ones are often designed to minimize the amount of wasted calculations. Given the variety of CMs, it is challenging, and still remains an open problem, to select the optimal CM for a given workload. This is primarily because the performance of CMs is sensitive to the workload as well as the underlying system platform.

We demonstrate the non-optimality of existing CM designs through experiments.



(a) Linked-List (x86)

(b) RB-Tree (x86)

(c) Skip-List (x86)

(d) Hash-Table (x86)

(e) Linked-List (powerpc)

(f) RB-Tree (powerpc)

(g) Skip-List (powerpc)

(h) Hash-Table (powerpc)

Figure 13: Comparison of different contention managers on different benchmarks and different platforms. (TinySTM, 16 threads)

We tested various CMs on both TinySTM [34] and RSTM [55, 56] distribution packages on two hardware platforms. The first platform is equipped with four 2.93GHz

quad-core Intel X7350 CPUs, and the other with one 3.0GHz quad-core IBM POWER7 CPUs where each core supports 4 hardware threads. We observed similar trends on both RSTM and TinySTM, and we only present the results of 16 threads on TinySTM in Figure 13 due to space limit.

Figure 13 illustrates that the performance of CMs varies with the benchmarks as well as the system platforms.

- *benchmark dependence*. This is observed on both platforms. For example, on the x86 platform, CMs with backoff (`AggressiveD`, `SuicideD`, `KarmaD`, and `TimestampD`) outperform the CMs without backoff on Linked-List and Skip-List. `Aggressive` performs the best on RB-Tree, but under-performs on Linked-List. Similar trends can also be observed on the POWER7 platform.

- *platform dependence*. For the same benchmark program, a CM may exhibit different performance characteristics across the platforms. For example, `SuicideD` performs best for Hash-table on POWER7, but is one of the worse CMs for the same benchmark on x86.

In summary, there does not exist a static choice of CM that can guarantee optimal performance. The results show that (a) the choice of CM has a significant impact on the performance of STM system (e.g. more than 4× performance difference was observed on RB-Tree as in Figure 13(f)); (b) the optimality of CMs depends on the workload (benchmark) and platforms; and (c) choosing a fixed CM will lead to significant performance variations when the workload or platform changes. Adaptive selection of an appropriate CM during run time is therefore essential to the performance of STM systems.

4.3 Profiling-based Adaptive Contention Management

To achieve adaptive contention management (ACM), we propose to periodically profile the CMs and dynamically select the one with the highest throughput. The whole

process is illustrated in Figure 14. At every profiling point, each CM in the pool (the selection of the CM pool will be discussed in Section 4.3.3) will be switched in and run for a period of time. The throughput of a CM is calculated upon the completion of its profiling. After the throughput values of all CMs are collected, the CM with the highest throughput will be selected for subsequent execution until the next profiling point.

As shown in Figure 14, an STM program may encounter multiple profiling points during its execution. The profiling interval (T) controls the frequency of profiling, and the profiling length (l_i for CM_i) affects the length of the profiling process. Profiling will inevitably cause overhead to the original program. Intuitively, the more profiling a program conducts (a smaller T or larger l_i), the more overhead will be incurred since sub-optimal CMs will be applied more often. However, if the profiling process is not frequent enough (T is too large), the ACM may not be responsive when the workload changes; furthermore, if the profiling process is not long enough (l_i is too small), then the profiling results may be inaccurate and may cause the system to choose a sub-optimal CM.

Because the optimal values of T and l_i are workload and platforms dependent, a major challenge in designing our profiling-based ACM is the optimization of the profiling interval T and profiling length l_i.

It is desirable to have $T \gg l_i$ since the objective of adaptation is to quickly select an optimal CM and then use it for the program execution. The consequence of selecting a sub-optimal CM is therefore expensive. The profiling accuracy should be prioritized over the profiling overhead. Therefore, in our method, we will start from a small profiling interval T that helps us quickly find for the minimum profiling length that satisfies the accuracy requirement, and then increase the profiling interval T to reduce the overhead incurred by unnecessary profiling.

The profiling overhead also depends on the selection of the candidate CMs. The

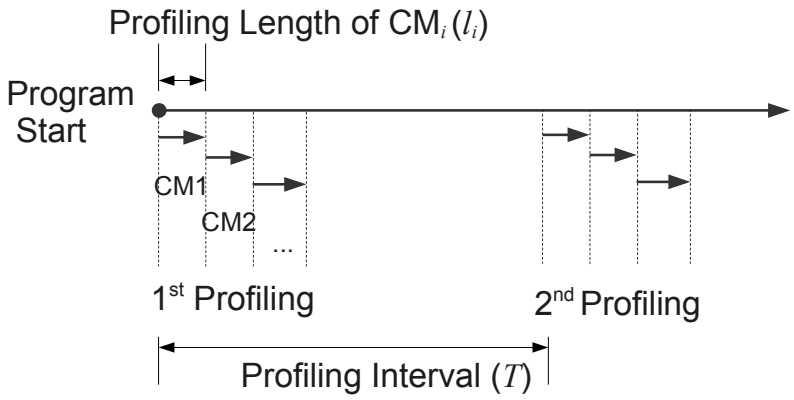


Figure 14: Periodic profiling process of CMs for the proposed adaptive ACM scheme

overhead consists of two parts: (1) the fixed overhead such as setting the system timer, switching between CMs, etc., and (2) during the profiling process, each CM will be tested for a certain period of time, sub-optimal CMs will lower the performance of the STM program. Part (1) of the cost is an implementation detail and is also related to STM designs. Experimental results suggest that this cost is marginal. Thus, we focuses on part (2) of the overhead. For notational convenience, we assume that we have k CMs in the pool. We use Th_i to denote the profiled throughput of a CM$_i$, and Th_{max} to denote the throughput of the optimal CM. We quantify the overhead by

$$O = \frac{\sum_{n=1}^{k}(Th_{max} - Th_i)l_i}{T} \tag{6}$$

which is the performance lost when profiling sub-optimal CMs.

Equation 6 indicates that the profiling overhead can be reduced by using a larger profiling interval T, minimizing the profiling length l_i for each CM, and carefully choosing the candidates to have a small number of candidates that tend to perform well (smaller k and $(Th_{max} - Th_i)$).

Next we discuss the three aspects of our design: Section 4.3.1 shows how we dynamically adjust profiling interval T and profiling lengths l_i to reduce unnecessary profiling; Section 4.3.2 depicts how we decide the profiling lengths to accommodate

all types of workload; And Section 4.3.3 discusses how to choose the candidate CMs.

4.3.1 Dynamic Adjustment of Profiling Interval and Profiling Length

The optimal values of the profiling interval and profiling length depend on various factors, and a major one is the characteristics of the workload. For example, a high-throughput STM program would require a shorter period of time for an accurate profiling. Workload with time-varying characteristics would require more frequent adaptation. Thus, fixed profiling interval and length as in [63] is undesirable.

In our proposed method, we dynamically adjust the profiling interval and length according to the workload. The profiling interval should be adjusted to the degree of time-variance of the workload. If the workload varies fast, the profiling interval needs to be shorter to be responsive. If the behavior of the workload is stable, we should extend the profiling interval. Similarly, the profiling length also needs to be dynamically adjusted so that it is long enough to ensure the profiling accuracy, but not so long to cause unnecessary profiling overhead.

It is expensive to verify whether a profiling result is accurate or not. For example, it is possible to track the standard deviation across all the profiling results. But this will require extra storage and computation. Note that the objective of profiling is not to track the precise throughput for each CM, but to identify which CM is better than others for the current workload and platform. We can therefore tolerate some profiling errors as long as they do not affect the comparison of the CMs. To balance the accuracy and overhead, in our adaptation scheme (shown in Algo. 5), we track the throughputs at two consecutive profiling points, and we consider the profiling results to be accurate if the difference is smaller than a threshold.

Algo. 5 shows our algorithm for the dynamic adjustment of the profiling frequency and length. At the end of the k^{th} profiling point, we compute the throughput for CM_i (Th_i^k) and compare it with the previous results Th_i^{k-1}. We use v_i to denote

the throughput variance between two consecutive profiling for CM_i (line 5). v_i is compared against threshold VAR_THRES to decide if the current profiling result is accurate or not. If not, the profiling length for this CM will be doubled at next profiling point (line 6).

We also record the accumulated variance *var* to detect changes in the workload. In line 10, we compare *var* with VAR_THRES × NB_CMS (where NB_CMS denotes the number of CMs in the pool). If *var* > VAR_THRES, it is indicative that the workload behavior has changed, so we will reset the profiling interval. In this case, T will be reset to an initial value (INITIAL_INT = 250 ms in our experiments). We are conservative in increasing the profiling interval and shrinking the profiling length, because the profiling accuracy should not be sacrificed for the overhead. Only when *var* < VAR_THRES (which means none of v_i is larger than VAR_THRES), we believe the profiling result has stabilized, so we double the value of T and cut l_i by half to reduce unnecessary profiling. In our design, T will be capped by INT_BOUND (set to 4 seconds in our experiments) to maintain the responsiveness of the profiling procedure.

4.3.2 Profiling Length

The profiling length l_i for each CM is another key design parameter. Two metrics can be used to time the profiling length: physical-time or logic-time. Frank in [63] choose to use the physical-time, which we call time-based profiling method (TPM). Instead, we can also use logic-time to measure the profiling length. In STM systems, commit and abort are two frequent and meaningful events, and are good candidates for tracking logic events. It is possible and convenient to profile each CM for a fixed amount of commits/aborts instead of time (l_i will be different for different CMs). We call these commit-based profiling method (CPM) and abort-based profiling method (APM) respectively. We next compare these three methods and show that APM is

Algo. 5 Adjustment of Profiling Interval and Profiling Length after the k^{th} Profiling Ends

Input:

Th_i^k: throughput of CM$_i$ in the k^{th} profiling.

Output:

l_i: profiling length for CM$_i$.

T: profiling interval.

```
1: var ← 0.0
2: for all cm do
3:    v_cm ← (|Th_cm^k − Th_cm^{k−1}|) ÷ Th_cm^k
4:    if v_cm > VAR_THRES then
5:       l_cm ← l_cm × 2
6:    end if
7:    var ← var + v_cm
8: end for
9: if var > VAR_THRES × NB_CMS then
10:   T ← INITIAL_INT
11: else
12:   if var < VAR_THRES and T < INT_BOUND then
13:      T ← T × 2
14:      for all cm do
15:         l_cm ← l_cm ÷ 2
16:      end for
17:   end if
18: end if
```

better than the other two.

We first quantify the performance overhead of the TPM, CPM and APM. As we showed in Equation. 6, the overhead is related to T, l_i and Th_i. Let us assume TPM will profile CM$_i$ for t_i seconds, thus we can replace l_i to t_i directly which represents the time spent by CM$_i$. The overhead of TPM can be calculated as

$$O_{TPM} = (\sum_{n=1}^{k}(Th_{max} - Th_i)t_i)/T \tag{7}$$

For CPM, we assume each CM is profiled for C_i commits, therefore l_i should be expanded to $\frac{C_i}{Th_i}$, and the overhead of CPM is

$$O_{CPM} = \sum_{i=1}^{k}(\frac{C_i \cdot Th_{max}}{Th_i} - C_i)/T \tag{8}$$

Similarly, if we assume each CM is profiled for A_i aborts, and the abort rate of CM_i is Ab_i, we can calculate the overhead of APM as

$$O_{APM} = (\sum_{n=1}^{k}(Th_{max} - Th_i)\frac{A_i}{Ab_i})/T \tag{9}$$

It can be seen from the equations that O_{TPM} is bounded if we set t_i to a small value (compared with T). For O_{CPM} and O_{APM}, because a CM may theoretically (and very rarely) take an arbitrary long period of time to commit or abort transactions, their values may be unbounded.

Although O_{TPM} can be bounded, it is very difficult to set the profiling length for TPM. This is because workload may vary significantly. For example, we observed over 100× variances in transaction throughput for benchmarks in the STAMP suites [25]. Given any profiling length, say 1 second, it may be appropriate for workload A, but insufficient for workload B. Note that our adaptation scheme adjusts the profiling length automatically, but setting the initial profiling length is still a challenge for TPM. Besides, for a workload with time-varying characteristics, transaction length may vary significantly as the program executes, parameter t_i has to be continuously adjusted, which will cause extra overhead (see experimental results in Section 4.5.3).

On the contrary, CPM and APM are both decoupled from physical-time, and do not have this drawback. Regardless of the transaction throughput of a workload, it will commit/abort transactions. We will be able to estimate the performance of the CM during the time period that certain number of commits/aborts occurred. For example, if we profile a CM for 1000 aborts, but see no commits, we can almost be sure that this CM is problematic. However, if we profile this CM for 1 second of time, and do not see any commits, we will not be able to tell whether this is a problem of the workload (transactions are too long) or a problem of the CM. CPM and APM are therefore more flexible choices than TPM.

In practice, O_{APM} rarely is unbounded. O_{APM} has a $Th_{max} - Th_i$ term on the

numerator. In practice, this term is small when the abort rate Ab_i is small: a CM with low abort rate tends to result in high transaction commit rate. More importantly, if a CM on the contrary generates a low transaction throughput Th_i and a low abort rate Ab_i at the same time, then it is likely that this CM is causing the program not to commit and not to abort, which is a sign of deadlocks. However, deadlocks are guaranteed not to occur in any properly designed STM systems [64] (This is actually a major advantage of STM). The overhead of APM is thus also bounded in practice.

Although properly designed STM systems can prevent deadlocks, under certain conditions, some CMs may still cause livelocks that results in a close to zero throughput Th_i. For example, in our experiments, `Aggressive` of RSTM with 16 threads on RB-Tree on x86 had a throughput of 44 transactions per second, while other CMs achieved more than 10^6 transactions per second. For such cases, O_{CPM} can be arbitrarily large (see experimental results in Section 4.5).

In terms of implementation cost, TPM is the highest. Because we only have one timer for both t_i and T in Linux, for each profiling process, TPM must adjust the interval of the timer at least twice for t_i and T respectively. Timer needs to be implemented through operating system support, which tends to be expensive. For CPM and APM, we can track the number of commits or aborts with a simple counter embedded in STM's commit or abort functions.

In summary, TPM has the advantage of bounded overhead, but it is inflexible in guaranteeing profiling accuracy and will cause higher implementation cost than CPM and APM. CPM and APM are more robust to variance in the workload with lower implementation overhead and better profiling accuracy, but CPM will be severely impacted if one of the candidate CM causes livelocks on the target workload and platform. APM is therefore better than the other two for measuring the profiling length. Experimental comparison of the three methods are presented in Section 4.5.

4.3.3 Selection of Candidate CMs

Selecting a proper pool of candidate CMs is also important for our design. An important design parameter for the pool selection problem is the size of the pool. A larger pool increases the probability of finding a better CM, at the cost of longer profiling period as well as higher implementation cost (e.g. memory storage). Our experimental results showed that 4 to 8 are reasonable sizes.

Another important factor is in the selection of individual CMs. Our experimental results show that there are two types of CMs: (1) those that perform well on some benchmarks (e.g. `Aggressive` on HashTable), but poorly on others (e.g. `Aggressive` on RB-Tree); (2) those that perform reasonably well across all workload and platforms, but may not be the best. Type 1 CM is preferred for our ACM because we can dynamically identify suitable CMs for a given workload.

4.4 Implementation

To thoroughly test the performance of the proposed method, we build our ACM scheme on both TinySTM and RSTM. These two STM systems follow two very different design logics while both exhibiting good transactional performance.

TinySTM is a word-based STM system developed by Felber et al. [34]. It is implemented in C, and the design target is to keep the code as simple and efficient as possible. Thus, it provides less configurations than RSTM to the programmers. In its write-back-ETL mode, it supports run-time switching of CMs (but requires programmers to specify which CMs to switch). Five basic CMs including `Aggressive`, `Suicide`, `SuicideD` (they call it `Delay`), `Karma` and `Timestamp` are shipped with distribution package, but other CMs can be easily plugged in because of its modular design. Note that for the backoff decision returned by a CM, TinySTM only accept the scheme that backoffs after a restart. The latest TinySTM version 1.0.0 is used in this Chapter.

```
1 stm_commit() {
2   ...
3   if (profiling) {
4     commits = ATOMIC_INC (&
          nb_commits[cur_CM]);
5 #ifdef _CPM_
6     if (commits > max_commits[cur_CM
          ] && thread_id == 0)
7       if (cur_CM == LAST_CM) {
8         profiling = False;
9         select_best_CM ();
10        adjust_prof_int_and_len ();
              /* Algo.5 */
11        set_next_profiling_time (&
              timer_handler);
12      } else
13        switch_cm ();
14 #endif
15  }
16  ...
17 }
```

```
19 stm_abort() {
20   ...
21 #ifdef _APM_
22   if (profiling) {
23     aborts = ATOMIC_INC (&nb_aborts[
          cur_CM]);
24     if (aborts > max_aborts[cur_CM]
          && thread_id == 0)
25       if (cur_CM == LAST_CM) {
26         profiling = False;
27         select_best_CM ();
28         adjust_prof_int_and_len ();
              /* Algo.5 */
29         set_next_profiling_time (&
              timer_handler);
30       } else
31         switch_cm ();
32   }
33 #endif
34   ...
35 }
```

```
37 stm_init() {
38   ...
39   cur_CM = 0;
40   reset_profiling_counters ();
41   profiling = True;
42 #ifdef _TPM_
43   set_next_switch_time (&
         timer_handler);
44 #endif
45   ...
46 }
```

```
47 timer_handler (){
48 #ifdef _TPM_
49   if (profiling) {
50     if (cur_CM == LAST_CM) {
51       profiling = False;
52       select_best_CM ();
53       adjust_prof_int_and_len ();
               /* Algo.5 */
54       set_next_profiling_time (&
             timer_handler);
55     } else
56       switch_cm ();
57   } else {
58     reset_profiling_counters ();
59     profiling = True;
60   }
61 #elif defined(_APM_) || defined(
       _CPM_)
62   if (!profiling) {
63     reset_profiling_counters ();
64     profiling = True;
65   set_next_switch_time (&
         timer_handler);
66   }
67 #endif
68 }
```

Figure 15: A snapshot of the added code of ACM for TinySTM

Differently, RSTM [23] is an object-based STM system implemented in C++. We use its release version 5. This version provides multiple choices for the configuration such as invisible-read or visible-read, lazy version management or eager version management, etc. Over 20 CMs are available in the package, though most of them are very similar. RSTM supports all three types of CM decisions including both backoff schemes, and every CM is implemented in a separate C++ class so that RSTM is almost compatible with any CM.

Our ACM can be considered as an add-on to the original STM system. By monitoring the run-time behavior of the workload, our ACM can adaptively adjust the current CM to maximize the overall performance. Figure 15 shows a snapshot of our modifications for TinySTM. The majority of our modification is in three STM interface functions, `stm_init`, `stm_commit`, and `stm_abort`, which exist for all STM systems. We implement our ACM with all three profiling methods, TPM, CPM and APM. Our design is easy to implement with less than 200 lines of code in total.

As shown in Figure 15, we use a flag `profiling` to denote if the program is currently being profiled, `cur_CM` to denote the current CM being used, and a counter `nb_commits` for each CM to record the number of commits. For APM, we also need one additional counter `nb_aborts` for each CM to record the number of aborts. Initially, when `stm_init` is called, we reset all the counters , set the current CM to the first one in the pool (`cur_CM=0`), and then set the profiling flag to True. If TPM is used, we also need to install a timer during `stm_init`. We use `setitimer(ITIMER_REAL, ...)` to set the timer. Upon the expiration of the timer, a SIGALRM will be delivered and our installed `timer_handler` function will be called. TPM will switch the CM in `timer_handler` function. For CPM and APM, CM switching is triggered in functions `stm_commit` and `stm_abort` respectively. When profiling completes for all the CMs, the best CM will be selected and the next profiling interval and length will be adjusted (line 8-11 and line 26-29).

4.5 Experimental Results

In this section, we present the experimental results and analysis. Two architectural platforms were used in the experiments:

- **x86_64**: The system is equipped with four 2.93GHz quad-core Intel X7350 CPUs and 128GB memory. Linux kernel version is 2.6.32, and gcc version 4.4.4 is used.

- **powerpc**: The system is equipped with one 3.0GHz quad-core IBM POWER7 CPU where each core supports four simultaneous hardware threads. 4GB memory is installed in the system. Linux kernel version is 2.6.32, and gcc version 4.4.4 is used.

We implemented our ACM scheme for both TinySTM and RSTM. TinySTM supports both x86 and powerpc platforms and works in 64-bit mode. We tested it on both platforms in the 64-bit mode. RSTM release 5 is 32-bit only and does not support Linux on powerpc systems. We tested it on the x86 platform using 32-bit mode.

Table 2: Summary of the tested CMs

CM	Description
`Aggressive`	always aborts the other transaction.
`Suicide`	always aborts self (called `Timid` in RSTM)
`Polite`	always backoffs before aborting the other transaction (not available in TinySTM)
`Karma`	always aborts the newer transaction (In RSTM, it will backoff before aborting self).
`Timestamp`	always aborts the less-productive transaction (In RSTM, it is called `Greedy`, and it will backoff before aborting self).
`AggressiveD`	`Aggressive` with backoff before a restart (called `AggressiveR` in RSTM).
`SuicideD`	`Suicide` with backoff before a restart (called `TimidR` in RSTM).
`PoliteR`	`Polite` with backoff before a restart (not available in TinySTM)
`KarmaD`	`Karma` with backff before a restart
`TimestampD`	`Timestamp` with backff before a restart

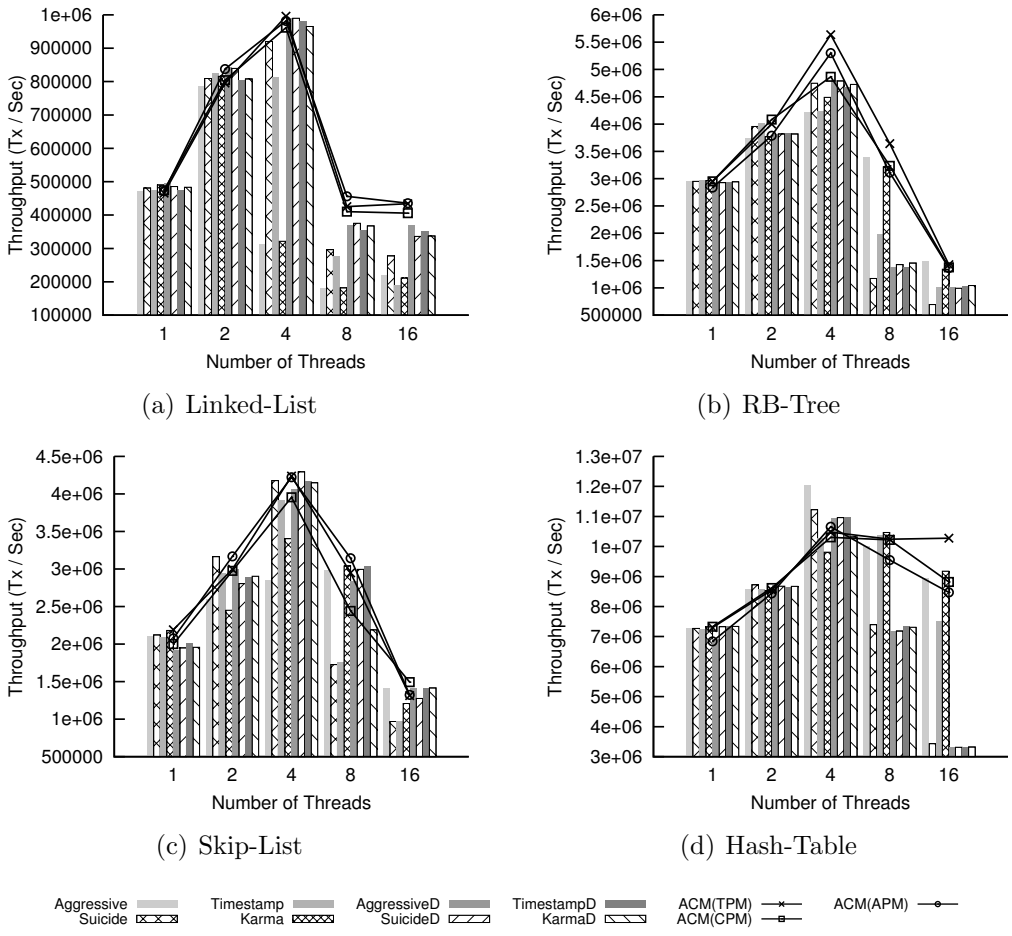
We selected four similar benchmarks from the TinySTM and RSTM. Linked-List, RB-Tree, Skip-List, and Hash-Table were chosen for TinySTM. Linked-List, RB-Tree, DList (Doubly Linked-List), and Hash-Table were chosen for RSTM. These

benchmarks cover a broad range of typical transactional workload. They differ in transaction size, transaction issue rate as well as the conflict rate. For example, the transaction sizes are random in Linked-List, but remain almost constant in Hash-Table. We use the default configuration for both TinySTM and RSTM, and each benchmark program was executed 10 seconds for each run. All the throughput values are averaged over five runs (we observed less than 10% variance).

Eight candidate CMs were chosen for both TinySTM and RSTM. This is a relatively large pool as we assume we have no *a priori* knowledge of the target workload and the STM system itself. In practice, if the STM designer knows which CMs are likely to be better in his/her system, CM candidate pool size can be reduced, which will reduce the overheads of our ACM scheme, and improve its performance as well. Table 2 lists the pool of candidate CMs.

For all the experiments, we used the same initial values to set up our ACM scheme. INITIAL_INT was set to 250 ms, INT_BOUND to 4 seconds, and VAR_THRES to 0.2. For the initial profiling length, we used 1 ms for TPM, 128 commits for CPM and 128 aborts for APM. The experimental results are presented in Figures 16, 17 and 18. It can be seen that on all benchmarks and platforms, the performance of CMs varied significantly. On x86 platform, the performance variance of CMs reached 40.5% for TinySTM (Hash-Table with 16 threads) and 86% for RSTM (RB-Tree with 16 threads). Similarly, on powerpc platform, this variance could be as high as 32% (Skip-List with 16 threads).

The proposed ACM schemes were able to adaptively choose an optimal CM during run time and consistently achieved performance that was close to the best static CM for all benchmarks and platforms. On some benchmarks, the performance of our ACM was even higher than that of the best CM in the candidate pool. For example, on x86 platform with TinySTM, our ACM(APM) outperformed the best static CM `SuicideD` by 18% for 8 and 16 threads on Linked-List(Figure 16(a)); on powerpc platform with



(a) Linked-List

(b) RB-Tree

(c) Skip-List

(d) Hash-Table

Figure 16: Performance comparison of ACM and static CMs on x86 platform for TinySTM.

TinySTM of 16 threads, ACM(APM) generated a 13% higher throughput than the best static CM `Timestamp` (Figure 17(c)). This is because these benchmarks had time-varying behavior during the execution. Any CMs in the pool, because they are static, would not be optimal throughout the execution of the benchmarks. On the contrary, our ACM scheme was capable of adapting the optimal choice of CMs during run time, and thus outperformed all the static CMs in the candidate pool.

Next we analyze and compare the three schemes of choosing profiling length, TPM, CPM and APM.

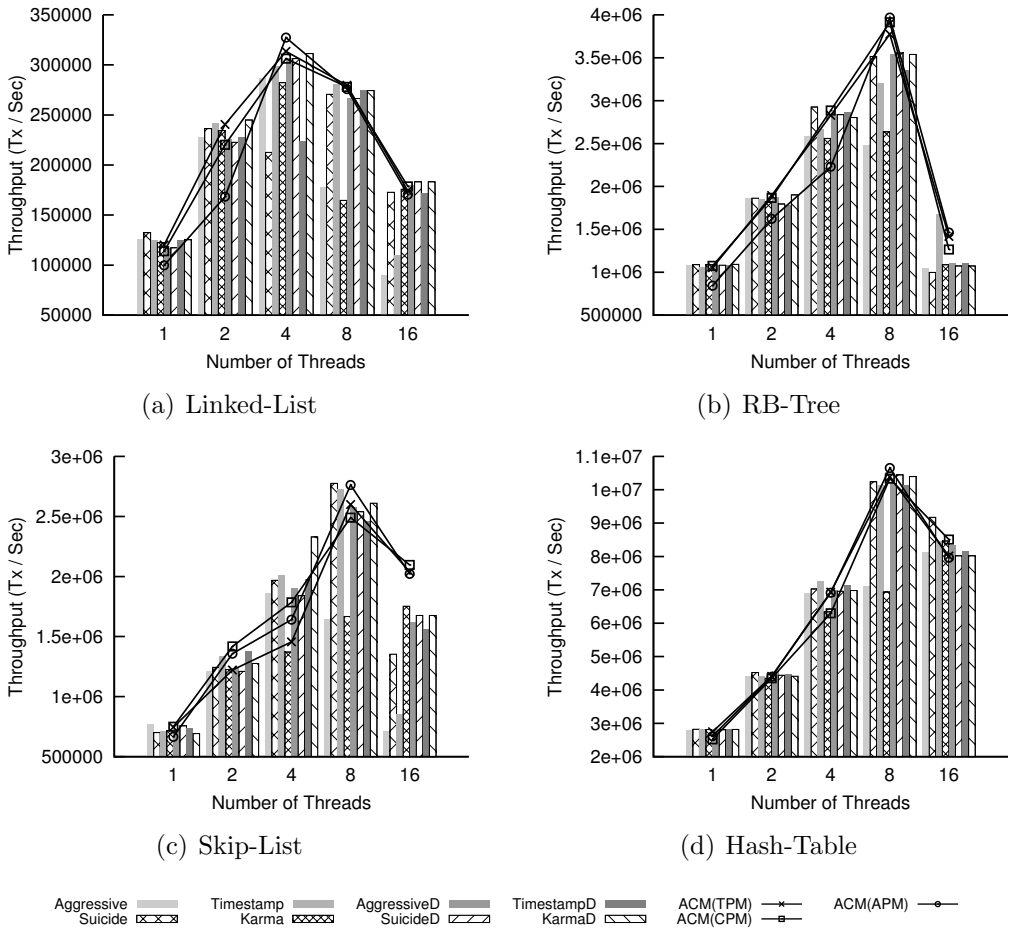


Figure 17: Performance comparison of ACM and static CMs on powerpc platform for TinySTM.

4.5.1 Implementation Overhead

As we discussed in Section 4.3.2, TPM has the implementation overhead that is mainly caused by frequently setting the timer. On the Hash-Table benchmark of RSTM (Figure 18(d)), all the candidate CMs had similar performance so that the adaptation frequently switched CMs. Moreover, because these CMs perform very stably, the profiling length l_i and profiling interval T are adjusted frequently (line 13-16 in Algo. 5. With TPM, each adjustment would require one extra timer re-installation. On high-throughput benchmarks such as Hash-Table, this overhead is magnified so that TPM performed worse than both CPM and APM (up to 55%).

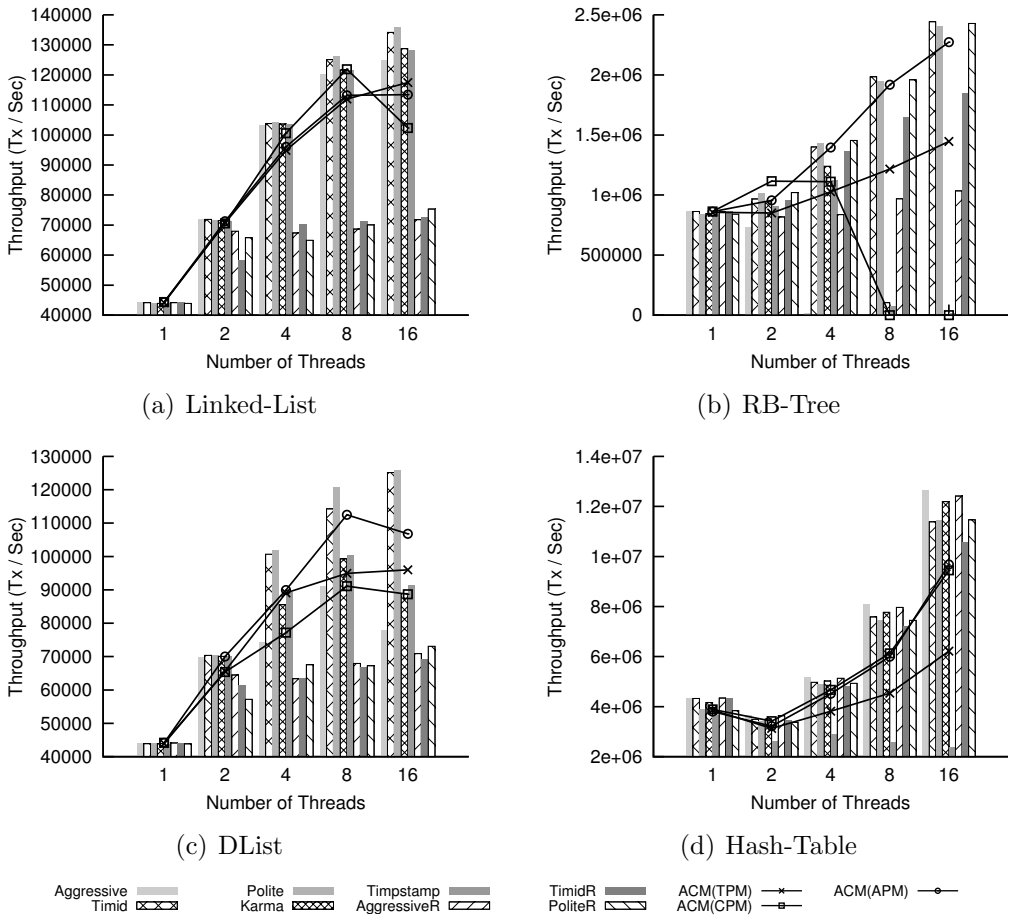


Figure 18: Performance comparison of ACM and static CMs on x86 platform for RSTM.

On the other hand, APM has one additional overhead than CPM which is the extra atomic operation in `stm_abort` (line 23 in Figure 15). This overhead is not obvious for most benchmarks, but on Hash-Table of TinySTM (Figure 16(d)), this overhead (extra 10^6 atomic operations per second) will cause APM to perform slightly worse than CPM by 4%. We could possibly use thread-local counters to avoid some atomic operations and thus reduce this overhead, but this is the implementation detail and not the focus of this Chapter.

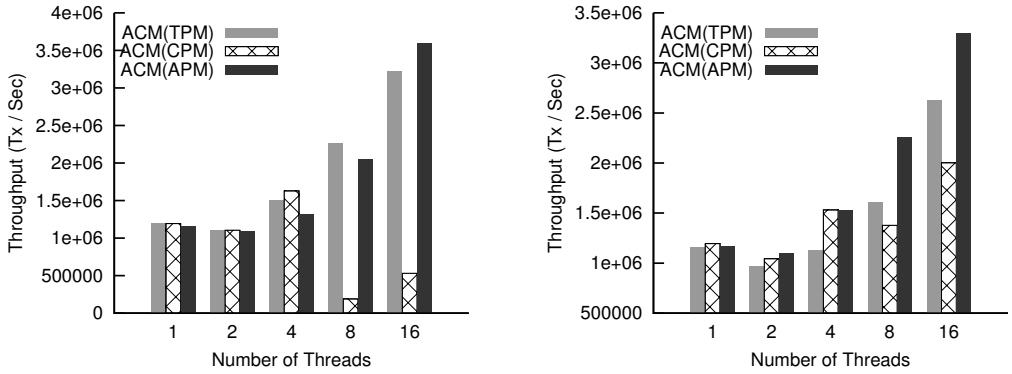
4.5.2 Livelock CMs

The RB-Tree on RSTM is an interesting case. Multiple CMs in the candidate pool caused livelocks. We observed numerous aborts but almost zero commit. The experimental results are demonstrated in Figure 18(b)).

For example, with 16 threads, `Aggressive` had only 44 commits per second and `Karma` had only 4853 commits per second, both of which were significantly lower than other benchmarks (more than 10^6 commits per second). When profiling these CMs, CPM will stall and wait for commits that were almost not occurring. CPM thus had a very low performance in this case (only 69 commits per second while TPM and APM were higher than 10^6). It is worth noting that TPM performed worse than APM by up to 36.6% when 16 threads were used. This was because TPM wasted certain amount of time profiling these livelock CMs. APM performed best for this scenario because the livelock CMs generated excessive aborts so that APM quickly collected enough aborts and switched to other well-performing CMs.

4.5.3 Time-varying Workloads

To demonstrate that the performance of TPM is workload sensitive, we synthesized a new benchmark for RSTM. This new benchmark integrated four types of workload Linked-List, RB-Tree, DList and Hash-Table from the benchmark suite of RSTM. The synthesized workload alternated through the above four types of workload, running each of them for a pre-set number of seconds. Figure 19 shows the results of two experiments. In both experiments, the new benchmark was run for 40 seconds in total. We set the switch time to 10 seconds in the first experiment and 2 in the second. TPM and APM exhibited similar performance, and CPM performed poorly on 8 and 16 threads. This is because some CMs will stall when RB-Tree is switched in for CPM. In the second experiment, when we decreased the switch time to 2 seconds, which simulated a more volatile workload that changes its behavior frequently. As shown



(a) Workload Switch Time: 10 seconds (b) Workload Switch Time: 2 seconds

Figure 19: Performance comparison of TPM, CPM and APM on the synthetic benchmark (RSTM, x86).

in Figure 19(b), APM outperformed TPM, and achieved a performance increase by 25% on 16 threads.

4.6 Summary

In this Chapter, we investigated a profiling-based adaptive contention management method for software transactional memory. We examined the performance of existing CM policies with a wide variety of benchmarks and platforms, from which we concluded that adaptation of CMs would be crucial to the performance of TM systems. We then presented our profiling based adaptive CM, and proposed to use logic-time (abort and commit events), instead of the physical-time, to determine the profiling length. We analyzed the profiling overhead for the proposed methods. We showed that the traditional physical-time based method is sensitive to workload and incapable of handling volatile workload. The experimental results validated our proposed method and showed that APM outperforms both TPM and CPM.

In addition to the adaptive contention management, our dynamic profiling framework for STM can also be used to tune other parameter for performance optimization such as the STM configurations (e.g. eager version management or lazy version management). We plan to investigate these extended usages of this profiling framework.

CHAPTER V

CONCLUSION

Multi-threaded computing becomes more and more popular and necessary, because we need efficient multi-threaded programs to unleash the power of current multi-/many-core processors. In multi-threaded programs, the synchronizations among the threads plays an important role. Improper uses of synchronization primitives would either result in incorrect program behaviors or cause excessive overhead. In this dissertation, we showed that the algorithm and the programming model are both essential for fast and correct synchronization in multi-threaded programs.

In Chapter 2, we show that a carefully-designed algorithm with a clear target for multi-threaded platform would reduce the needs of synchronization so that the efficiency of its multi-threaded implementation is improved considerably. We target on the max-flow problem and design an asynchronous algorithm. The algorithm is based on the classical push-relabel algorithm, which is essentially sequential and requires intensive and costly lock usages to parallelize. The newly designed push and relabel operations are executed completely asynchronously and each individual process/thread independently decides when to terminate itself. We further propose an asynchronous global relabeling heuristic to speed up the algorithm. We prove that our algorithm finds a maximum flow with $O(|V|^2|E|)$ operations where the $|V|$ is number of vertices and the $|E|$ is the number of edges in the network. We also prove the correctness of the relabeling heuristic. Extensive experiments show that our algorithm exhibits close-to-linear scalability as the number of processor cores increases and out-performs the lock-based parallel push-relabel algorithm by up to 49% in the execution time.

However, designing such a new algorithm requires not only large efforts, but also fortunes. For the existing algorithms that still need synchronizations, we have to improve the programming model to provide them a fast and convenient synchronization method. Therefore, in Chapter 3 and Chapter 4, we study the transactional memory, a promising synchronization paradigm to replace locks. Compared with lock-based schemes, transactional memory is expected to significantly reduce the difficulties of parallel programming and debugging, the vulnerability to failures and faults, and the likelihood of deadlocks. Thus, it provides a better programmability over the locks.

Chapter 3 present a queuing-theory-based analytical model to evaluate the performance of transactional memory. Based on the statistical characteristics observed on actual experiments, we model each transaction as a client requesting services from the computing system. Continuous time Markov chain is used to describe the start and completion (commit or abort) of the transactions. Experimental results show that our model predicts the performance of real transactional memory systems with an average error rate of 7.9%.

This performance model theoretically reveals that the contention level of transactional memory systems significantly affects its performance. Therefore, Chapter 4 presents an adaptive contention management scheme that aims to reduce the contention level for software transactional memory systems for any workload and platform. We show that adaptive contention management is necessary and feasible. We introduce a profiling-based method that would choose a suitable CM for a given workload and system platform during run-time. We also propose to use logic-time (transactional commit or abort events) to measure the profiling length and compare it with the traditional physical-time-based method. Experimental results demonstrate that our propose adaptive contention manager outperforms static contention manager across benchmarks and platforms.

From the above research, we show that it is necessary and worthwhile to explore

both the algorithm design aspect and the programming model aspect for multi-thread computing. New algorithm designs with the target of the multi-threading would benefit from the emerging multi-/many-core platform. At the same time, we still need to improve the current programming model to enhance the programmability and performance for the algorithms that rely on intensive synchronizations.

REFERENCES

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 289–300.

[2] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[3] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?" *Queue*, vol. 6, no. 5, pp. 46–58, 2008.

[4] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui, "Why stm can be more than a research toy," *Commun. ACM*, vol. 54, pp. 70–77, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1924421.1924440

[5] R. J. Anderson and a. C. S. Jo "On the parallel implementation of goldberg's maximum flow algorithm," in *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1992, pp. 168–177.

[6] D. Bader and V. Sachdeva, "A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic," in *PDCS '05: Proceedings of the 18th ISCA International Conference on Parallel and Distributed Computing Systems*, 2005.

[7] A. V. Goldberg, "Recent developments in maximum flow algorithms (invited lecture)," in *SWAT '98: Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*. London, UK: Springer-Verlag, 1998, pp. 1–10.

[8] D. S. Johnson and C. C. McGeoch, Eds., *Network Flows and Matching: First DIMACS Implementation Challenge*. Boston, MA, USA: American Mathematical Society, 1993.

[9] L. R. Ford and D. R. Fulkerson, *Flows in Networks*. Princeton University Press, 1962.

[10] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, 1972.

[11] E. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Mathematics Doklady*, vol. 11, pp. 1277–1280, 1970.

[12] A. V. Karzanov, "Determining the maximal flow in a network by the method of preflows," *Soviet Mathematics Doklady*, vol. 15, pp. 434–437, 1974.

[13] H. N. Gabow, "Scaling algorithms for network problems," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 148–168, 1985.

[14] A. V. Goldberg and R. E. Tarjan, "Finding minimum-cost circulations by successive approximation," *Math. Oper. Res.*, vol. 15, no. 3, pp. 430–466, 1990.

[15] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," in *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1986, pp. 136–146.

[16] Y. Shiloach and U. Vishkin, "An o(n^2 log n) parallel max-flow algorithm," *J. Algorithms*, vol. 3, no. 2, pp. 128–146, 1982.

[17] J. JáJá, *An introduction to parallel algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1992.

[18] D. Culler, J. P. Singh, and A. Gupta., *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998.

[19] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms, 2nd Edition*. MIT Press, 2001.

[20] D. D. K. Sleator, "An o(nm log n) algorithm for maximum network flow," Ph.D. dissertation, Stanford University, Stanford, CA, USA, 1981.

[21] J. Cheriyan and K. Mehlhorn, "An analysis of the highest-level selection rule in the preflow-push max-flow algorithm," *Information Processing Letters*, vol. 69, pp. 69–239, 1998.

[22] Z. He and B. Hong, "Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –10.

[23] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the overhead of software transactional memory," Computer Science Department, University of Rochester, Tech. Rep. TR 893, Mar 2006.

[24] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Jun 2007.

[25] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[26] Z. He and B. Hong, "On the performance of commit-time-locking based software transactional memory," *The 11th IEEE International Conference on High Performance Computing and Communications (HPCC-09)*, 2009.

[27] Z. He and B. Hong, "Impact of early abort mechanisms on lock-based software transactional memory," *The 16th IEEE International Conference on High Performance Computing (HiPC-09)*, 2009.

[28] J. Gray, "The transaction concept: virtues and limitations (invited paper)," in *VLDB '81: Proceedings of the seventh international conference on Very Large Data Bases*. VLDB Endowment, 1981, pp. 144–154.

[29] T. Knight, "An architecture for mostly functional languages," in *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM Press, 1986, pp. 105–112.

[30] N. Shavit and D. Touitou, "Software transactional memory," *Journal of Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[31] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2005, pp. 48–60.

[32] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*. IEEE Computer Society, Jun 2004, p. 102.

[33] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2009, pp. 155–165.

[34] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 237–246.

[35] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: Log-based transactional memory," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb 2006, pp. 254–265.

[36] M. Herlihy, M. Moir, and V. Luchangco, "A flexible framework for implementing software transactional memory," in *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, oct 2006, pp. 253–262.

[37] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. IEEE Computer Society, Jun 2005, pp. 494–505.

[38] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, "Metatm/txlinux: transactional memory for an operating system," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2007, pp. 92–103.

[39] S. Dolev, D. Hendler, and A. Suissa, "Car-stm: scheduling-based collision avoidance and resolution for software transactional memory," in *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, August 2008, pp. 125–134.

[40] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," *IEEE Micro*, vol. 26, no. 1, pp. 59–69, 2006.

[41] L. Baugh, N. Neelakantam, and C. Zilles, "Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, June 2008.

[42] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *Proceedings of Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

[43] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 336–346.

[44] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, "Architectural support for software transactional memory," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 185–196.

[45] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott, "An integrated hardware-software approach to flexible transactional memory," in *Proceedings of the 34rd Annual International Symposium on Computer Architecture*, Jun 2007.

[46] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "Eazyhtm: eager-lazy hardware transactional memory," in *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2009, pp. 145–155.

[47] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[48] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[49] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Logtm-se: Decoupling hardware transactional memory from caches," in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture(HPCA)*, Feb 2007.

[50] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, "Software transactional memory for dynamic-sized data structures," in *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing.* New York, NY, USA: ACM, 2003, pp. 92–101.

[51] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Object-Oriented Programming, Systems, Languages, and Applications*, Oct 2003, pp. 388–402.

[52] R. Ennals, "Efficient software transactional memory," Intel Research Cambridge Tech Report, Tech. Rep. IRC-TR-05-051, Jan 2005.

[53] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: a high performance software transactional memory system for a multi-core runtime," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming.* New York, NY, USA: ACM, 2006, pp. 187–197.

[54] H. Avni and N. Shavit, "Maintaining consistent transactional states without a global clock," in *SIROCCO '08: Proceedings of the 15th international colloquium on Structural Information and Communication Complexity.* Berlin, Heidelberg: Springer-Verlag, 2008, pp. 131–140.

[55] W. N. Scherer, III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing.* New York, NY, USA: ACM, 2005, pp. 240–248.

[56] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, "A comprehensive strategy for contention management in software transactional memory," *SIGPLAN Not.*, vol. 44, no. 4, pp. 141–150, 2009.

[57] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott, "Alert-on-update: a communication aid for shared memory multiprocessors," in *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming.* New York, NY, USA: ACM, 2007, pp. 132–133.

[58] A. Heindl and G. Pokam, "An analytic framework for performance modeling of software transactional memory," *Comput. Netw.*, vol. 53, no. 8, pp. 1202–1214, 2009.

[59] D. E. Porter and E. Witchel, "Modeling transactional memory workload performance," in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2010, pp. 349–350.

[60] Z. He and B. Hong, "Modeling the run-time behavior of transactional memory," *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, vol. 0, pp. 307–315, 2010.

[61] R. Guerraoui, M. Herlihy, and B. Pochon, "Polymorphic contention management," in *Proceedings of the 19th International Symposium on Distributed Computing (DISC 2005)*. LNCS, Springer, 2005, pp. 26–29.

[62] T. Heber, D. Hendler, and A. Suissa, "On the impact of serializing contention management on stm performance," in *OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 225–239.

[63] J. Frank and R. Chun, "Adaptive software transactional memory: A dynamic approach to contention management," in *PDPTA*, H. R. Arabnia and Y. Mun, Eds. CSREA Press, 2008, pp. 40–46.

[64] T. Harris, A. Cristal, O. S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero, "Transactional memory: An overview," *IEEE Micro*, vol. 27, no. 3, pp. 8–29, 2007.